CONSISTENT ACCESS TO REPLICATED WEB SERVICE REGISTRIES

MAHANTESH SURGIHALLI

Consistent Access to Replicated Web Service Registries

by

©Mahantesh Surgihalli

A thesis submitted to the
School of Graduate Studies
in partial fulfillment of the
requirements for the degree of
Master of Science

Department of Computer Science
Memorial University of Newfoundland

October 2006

St. John's Newfoundland

Your file Votre référence
ISBN: 978-0-494-30509-6
Our file Notre référence
ISBN: 978-0-494-30509-6

Abstract

Web Services is supposed to be the next biggest revolution in IT industry as it has created a platform-neutral environment for business processes to communicate. Because of the interoperability provided by Web Services, the number of companies using it is expected to grow exponentially. As such, the number of queries to name directory services for Web Services, Universal Description Discovery and Integration (UDDI) registries, is expected to be very high. To facilitate high availability, UDDI version 3 suggests replication of data in the registry.

For quick response and scalability, a lazy replication scheme is appropriate for UDDI registries. Here, update transactions are executed at primary node(s) as primary transactions, committed, and then updates are transmitted to other nodes asynchronously, which are then executed at those nodes as refresh transactions. Then, due to the updates at different nodes being done at different times, a user may read a recent version of a data item at one node and, later, read an older version of the same or some other related data item at some other node. As a result, the user may obtain an inconsistent view of the registry. For a user accessing different nodes in operations of a session, a session guarantee mechanism is required to ensure a consistent view of the registry.

In this work, we propose a transaction execution protocol that (i) uses a lazy replication scheme, (ii) ensures one copy serializability, and (iii) incorporates a fine grained session guarantee mechanism. Transactions are classified as update or read only transactions and also as local or global transactions. We extensively study these transaction types and design a

protocol that exploits the compatibility among them to achieve high performance. The protocol is an extension of two phase locking protocol which has flexibility to execute a transaction pessimistically or optimistically. Message propagation mechanism is designed such that updates of conflicting transactions (only) are delivered causally.

The protocol is designed first for a fully replicated system, and then extended to the partially replicated system. Lastly, we propose a deadlock detection and resolution mechanism.

Acknowledgements

The first acknowledgement in every thesis goes to the supervisor. My thesis supervisor Prof. K Vidyasankar deserves special acknowledgement not only for his guidance and support but also for identifying my strength and weakness in my technical skills and providing clear and efficient feedback. I would like to heartily thank him for spending millions of minutes of his time on my thesis during the course of my program.

I would like to thank my parents Chandrashekarappa, and Ratnamma, for their love, encouragement, and constant support. I thank my sisters Veena, Aruna, and Vijaya for always being there for me. I would also like to thank my uncle Dr. H. B. Chandrasekhar, and Umakka for giving a new direction in my life and always suggesting improvements in me. Special thanks to Usha Vidyasankar for being kind and helpful.

This acknowledgement is incomplete without the mention of my dearest friend, Debmalya Biswas, with whom I used to discuss almost everything under the Sun. Thanks for having numerous technical discussions both online and offline. I am thankful to Dr. Ananthanarayana for many stimulating discussion we had during his stay in St John's. I also thank to my friends Joseph, Donald, Ram, Manoj, Koshi, Prem, Jianmin Su, Chen, and Pradeep (and the list goes on) for the good time I shared with them during my program. I am also thankful to all my friends in Pantaru group who helped me to keep in touch with my motherland.

I would like to thank everyone in the Computer Science Department at Memorial University for providing the perfect study environment, especially, Dr. Wolfgang Banzhaf, Ms. Radha

Gupta, and Ms. Elaine Boone for their encouragement and support. Last but not the least, I would like to thank the various sources within and outside the University for sponsoring my studies and participation at PDCS 2005 and WISE 2005.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Web Services

The World Wide Web (WWW) has revolutionized the field of Information Technology (IT). The WWW has changed the way people think and communicate. Web Services is expected to be the next big revolution in IT. The WWW boom was basically due to its capability to communicate between an application and its users. The hype around Web Services is due to its promise of enabling the communication between the applications without user intervention, because of which companies can integrate and reuse software that they or others have already built, historically an expensive and time-consuming process. Although communication between applications was possible in traditional frameworks, such as CORBA [CORBA] and DCOM [DCOM], Web Services is considered a landmark technology because of its standardization and wide acceptance in the IT community. The World Wide Web Consortium (W3C) defines Web Services as "a software application identified by a Uniform Resource Identifier (URI), whose interfaces and bindings are capable of being defined, described, and discovered as Extensible Markup Language (XML) artifacts. A Web Service supports direct interactions with other software agents using XML-based messages via Internet-based protocols". Service Oriented Architecture (SOA) has fueled the growth of Web Services by enabling dynamic discovery and usage over the Internet.

Usually, a Web Service has the following properties:

- Service Description: It explains what a Web Service can do. The public interface is published along with the Web Service, so that a user knows how to invoke it. This description should be at least human-readable so that a developer can integrate the Web Service. Typically, this is machine-readable using W3C's Web Service Description Language (WSDL) [WSDL]. WSDL uses XML grammar to identify all public methods, method arguments and return values.

- Service Discovery: The consumers of Web Services go to the service broker to find one of the providers through a static or dynamic brokerage system. This brokerage system is basically a name service registry. Universal Description, Discovery, and Integration (UDDI) [UDDI] is the standard XML based registry for Web Services.

- Service Interactions: Simple Object Access Protocol (SOAP, although this full name has been dropped in Version 1.2 of SOAP specification) [SOAP] is the de-facto standard for information exchange between Web Services. Initially, SOAP was designed as remote procedure call over HTTP. Nevertheless, SOAP can be used in a variety of messaging systems and can be delivered via a variety of transport protocols. The major difference between SOAP and other frameworks, such as CORBA and DCOM, is that SOAP is language and platform independent (whereas others are not).

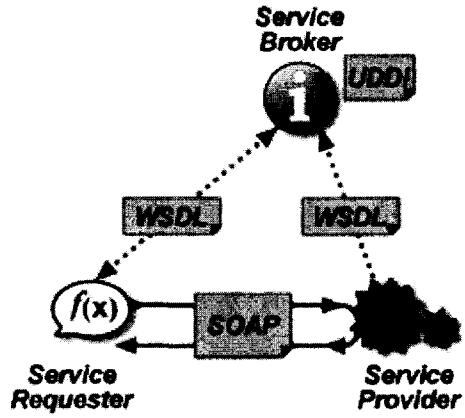

Figure 1.1 Web Services usage scenario*

The Web Services usage scenario can be explained with Figure 1.1 as follows:

1. The service provider prepares a WSDL document describing the services it provides. The provider publishes (registers) the WSDL document with an UDDI registry.
2. The client queries the UDDI registry. The registry returns not only descriptive information about the service provider but also information regarding where and how the service can be invoked.
3. The client interacts with the provider using the above information.

Let us consider an example scenario where the Web Services technology can be used.

Example 1.1: Let us consider a trusted environment where every business organization trusts every other organization. Consider such organizations to be Intel,

* http://en.wikipedia.org/wiki/Web_service

Microsoft, and Apple Computers. Let us assume that Intel manufactures a new hardware device. Microsoft releases the operating system for Intel's hardware. Apple Computers releases the device driver, a software which can be used by the old legacy system. When the hardware device is deployed to the user, there may be incompatibility problems which are traditionally very difficult for end users to resolve. Even intermediate users may have difficulty in finding out if the problem is with the hardware, operating system, or device driver.

Web Services technology can solve this problem. Whenever there is such an incompatibility, the software stops executing the module and an exception is reported. Each of the modules is associated with a category. The program now queries the UDDI registry for Web Services using the category of the module. Once the search is successful, it invokes the Web Service at the provider site. The solution for the problem is dependent on service selection and its invocation at the provider's site. That is, if the provider is Intel, it may replace the hardware. On the other hand, if the provider is Microsoft or Apple Computers, they may install an update to the existing software. This kind of automation enables even a novice user to handle the latest hardware without worrying much about the internal details. This kind of automation is not possible in other name service directories, such as ebXML [ebXML], as they allow users to define their own data structure. The strength of the UDDI data model lies in its precise definition of all its entity types.

1.2 UDDI and Replication

UDDI is commonly regarded as a cornerstone of the Web Services paradigm. UDDI registries are accessed by providers who publish Web Services, requesters who look for Web Services and by other registries that need to exchange information. The customers of UDDI use APIs for interacting with the registry. APIs in UDDI can be classified into Publish and Inquiry Application Programming Interfaces (APIs) which are used by publisher of service and all customers, respectively. As only publishers use publish API and all the consumers use inquiry API, the number of queries is expected to be very high. In order to provide high availability and fault tolerance, version 3 of UDDI has moved to replicated UDDI registries, with full replication. The replication process imposes the overhead of maintaining consistency among replica copies. Although consistency issues have been considered in the conventional databases, they do not consider the configuration as specified in the UDDI specification. As the number of queries to UDDI registries is expected to be very high, accessing UDDI is likely to become bottleneck.

The main requirements of an UDDI registry are high throughput, low response time, high availability and accurate access to the entries in it. As UDDI is still evolving, consistency issues are still unaddressed.

1.3 Objectives of the thesis

Web Services is considered as the next big wave in the field of the Information Technology because of its interoperability with heterogeneity. As the user's dependency on Web Services increases, UDDI registries, the name directory services for Web Services are likely to become the bottleneck for their usage. UDDI version 3 has adopted replication of data items in an effort to solve this problem. In this thesis we develop the following subsystem to address the consistency issues in UDDI:

1. Replication protocol for fully replicated registries: The most commonly accepted correctness criterion in replicated database systems is one copy serializability, in which a user sees the entire system as a one copy system. As tight synchronization among nodes for replication of data items is not suitable for UDDI registries, a lazy replication scheme [HSAE03, DS04, ATSGB05] (in which data items are updated first at one node and the updates are propagated later to other nodes) is appropriate. However, in this system, as the updates at different nodes are done at different times, a user may read recent version of a data item at one node and later may read an older version of the data item at another node. As a result, the user may obtain an inconsistent view of the system. To avoid this, that is, to ensure a consistent view of the system, a session guarantee mechanism is required.

   The two phase locking protocol is the basic component of the replication protocol. If a transaction has at least one write operation, it is called an *update* transaction;

otherwise, it is a *read-only* transaction. We also classify transactions as *local* or *global*. We propose a two-phase locking protocol that achieves high performance based on the compatibility among different types of transactions. The protocol has flexibility to execute a transaction, either pessimistically or optimistically. It also provides a fine grained session guarantee. This part of the work appears in the proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2005), Phoenix, USA, November 2005.

2. Replication protocol for partially replicated registries: In partially replicated systems, updates of a data item need to be sent only to those nodes which have a copy of that data item. Therefore, for enhanced performance, a multicast mechanism is appropriate, instead of the broadcast mechanism that is used in fully replicated systems. The replication protocol and the session guarantee mechanism are extended to suit this requirement of partially replicated systems.

3. Deadlock resolution mechanism: Distributed deadlocks are possible as the replication protocol uses non-conservative locking mechanism. A mechanism that handles deadlocks, by means of detection and resolution is proposed. The advantages of the mechanism include resolution of complex configuration of deadlocks deterministically and non-abortion of transactions due to deadlocks which do not exist at the point of resolution.

4. Session guarantee mechanism: The session guarantee mechanism is designed to ensure a serialized view of transaction's updates executed in a session if these transactions conflict with each other and, in addition, ensures a consistent view of the operations of a service provider's session.

1.4 Structure of the thesis

The rest of the thesis is structured as follows. Chapter 2 presents a brief overview of the challenges and related work in conventional replicated databases and session guarantees. In Chapter 3, we introduce the architecture of UDDI. In the transactional framework for replication, we classify the transactions based on their operations and execution location. Then, we consider session guarantees in a lazy replicated registry. Chapter 4 deals with the replication protocol. We also provide the mechanisms for message propagation and session guarantees. A correctness proof is given. A protocol for partial replication is considered in Chapter 5. This protocol is an extension of the replication protocol in Chapter 4. In Chapter 6, we give an algorithm to detect and resolve distributed deadlocks. The algorithm is capable of detecting complex configurations of deadlock cycles. Deadlocks are resolved by aborting a transaction in the cycle. In Chapter 7, we extend the session guarantee mechanism discussed in chapter 4 to UDDI context. Chapter 8 concludes the work and provides directions for the future work.

Chapter 2

Background

In this chapter, we first explain the basic concepts of UDDI and then the replication protocol.

2.1 UDDI

Unlike ebXML [ebXML], UDDI does not allow users to define their own data model. UDDI defines data structures and API's for publishing service descriptions in the registry (publish API) and querying the registry to look for published descriptions (inquiry API).

An UDDI information model is composed of instances of the following entity types (refer Figure 2.1). Their descriptions are as follows:

1. BusinessEntity (BE): Business or organization that typically provides Web Services.
2. BusinessService (BS): A collection of related Web Services offered by an organization described by a BE.
3. BindingTemplate (BT): Technical information necessary to use a particular Web Service.

4. tModel (tM): The cryptic name stands for "technical model", representing a reusable concept, such as a Web Service type, a protocol used by Web Services, or a category system.

5. PublisherAssertion: Describes, with respect to a BE, the relationship the BE has with another BE.

6. Subscription: Describes a standing request to keep track of changes to the entities described by the subscription.

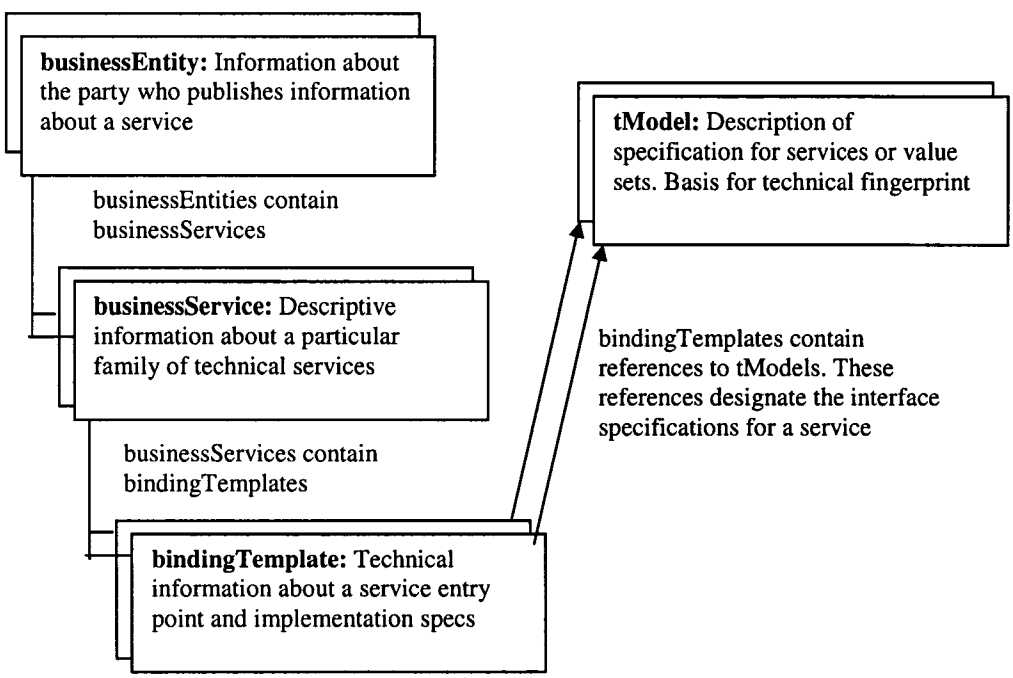

Figure 2.1 Core UDDI data structure

It is easy to see from Figure 2.1 that one BE may contain multiple BS. Similarly, one BS may contain multiple BT. Thus, a container relationship exists among these entities. But BT has an external link to the tM. So, there is a reference relationship between these two. Note that UDDI (or the UDDI specification) only defines logical organization and is independent of how they are stored physically.

The UDDI registry has the following components:

1. White Pages: Contains address, contact, and known identifiers of a business.
2. Yellow Pages: Contains industrial categorizations based on standard taxonomies for a business.
3. Green Pages: Contains technical information about services exposed by the business.

2.2 Replication

Replication of data items is one of the most common methods to increase the availabily and reliability of a system. Replication increases the availability by facilitating reading of a data item from a local node instead of a remote node. Reliability is ensured by redundancy, that is, if one of the nodes fails, another can take over and still keep the system running.

2.2.1 Replica Control

The price paid for providing availability and reliability is keeping all the replicas consistent. That is, whenever there is an update on any one of the data items, it has to be propagated to all the nodes. Maintaining all the nodes consistent is the aim of replica control. Replica control has been studied extensively in conventional databases. In a replicated database scenario, the most commonly accepted correctness criterion is *one copy serializability* (1SR) [BHG87]. To a user, the replicated database system appears as a single non-replicated database system. This correctness criterion is the strongest known, as it ensures commitment of a transaction at all the nodes if it commits at any one node, and all of the nodes reach the same consistent state. Other weaker correctness criteria, such as eventual consistency, allow concurrent conflicting transactions at different nodes to commit at the nodes they are submitted. The execution may not be serializable. Later, reconciliation operations [S84, FM82, X84] are performed to bring the database to a consistent state. The database replication issues can be divided into two parts — execution of transactions and propagation of updates between nodes. The transaction manager (TM) takes care of execution of transactions and the broadcast primitives take care of propagation of updates.

The two main classes of replica control strategies are *lazy* [HSAE03, DS04, ATSGB05] and *eager* [KA98, KA00]. In eager replication, for the execution of a transaction there is communication between all the nodes. Only after coordinating with these nodes does the transaction commit. The user obtains a response from the

system only after the transaction has successfully committed at all those nodes. In lazy replication, a transaction is first executed and committed at one node called the *primary* node and updates are propagated later to the other nodes called *secondary* nodes, asynchronously. Transactions executing at the primary and secondary nodes are called primary transactions and refresh transactions, respectively. As soon as the primary transaction is executed, the user obtains the response from the system.

2.2.2 Eager Replication

Eager replication essentially implements the Two Phase Commit protocol (2PC) [BHG87]. After executing a transaction, one of the nodes is selected as a coordinator which initiates the voting phase. All the nodes participate in the voting phase. The coordinator based on the votes executes a decision in the decision phase. As per the decision, the transaction commits at all nodes or does not commit at any node. The value for a data item at all the nodes will be the same at any point in time. There is no uncertainty period where different nodes have different values for a data item. Hence, this method is easy to facilitate to the user a notion of a single logical copy of the database.

2.2.3 Lazy Replication

2PC, which is a part of the eager replication, is often not feasible for real life applications, as it requires a lot of coordination among the nodes. [GHOS96] shows that as one node is added into the replicated system, a ten times increase in coordination is required which causes a thousand times increase in deadlock or

reconciliation rates. This scalability problem can be solved by relaxing the atomicity requirement of the eager replication. As a result, replica nodes may not be mutually consistent. In lazy replication, the need for the atomic commitment protocol is relaxed. It increases the availability by decreasing the response time of the replicated system. This also makes the system scalable with the increase in the number of nodes, as it decouples the primary transaction execution and propagation of updates to other nodes. Therefore, the lazy replication scheme is popular among commercially available databases. UDDI has used the lazy replication strategy [UDDI]. Inconsistency arises in the lazy replication as replicated nodes may be out of synchronization for a certain period of time. A user interacting with the system with a sequence of transactions may obtain an inconsistent view if different replica nodes are accessed over a period of time. A session guarantee mechanism aims to give a consistent view of the replicated system to each user, individually.

2.2.4 Session Guarantee

"A session is an abstraction for a sequence of read and write operations performed during execution of an application" [TDPSTW94]. The lazy replication may create multiple stale versions of the same data item at different nodes. A user is not aware of an internal organization of the replicated database and to him the system appears as a single copy database. The user's operations in a session is scheduled by the system such that the first operation reads a data item at a node, and a later operation reads the same data item from another node which is stale as compared to the previous

operation. This is an inconsistent view to the user. The session guarantee mechanism is tailored to provide a consistent view of the system on a per user basis. Different types of session guarantees are described in [TDPSTW94] as follows:

1. Read Your Writes: If any read R operation follows write W operation in the same session where both operations are performed on the same data item, then R should read the updates of at least W (or higher versions).

   Example 2.1: If W(b) is performed at Node-X, then R(b) of the same session cannot be executed at Node-Y until the update of W(b) is incorporated at that node.

2. Monotonic Reads: If read R_1 occurs before read R_2 in the same session and R_1 accesses Node-X at time t_1 and R_2 accesses Node-Y at time t_2, then R_2 should at least see the database state which was present when R_1 was executed at Node-X.

   Example 2.2: Let $R_1(b)$ be executed at Node-X which has already executed a set of write operations, namely W_1, W_2, W_3,...,W_n. Later at Node-Y, $R_2(c)$ of the same session cannot be executed until changes of W_1, W_2, W_3,...,W_n have been incorporated at that node.

3. Writes Follow Reads: At every node, writes made during the session are ordered after any writes whose effects were seen by previous reads in the same session.

   Example 2.3: Let $R_1(b)$ occur before $W_2(c)$ at Node-X in the same session. R_1 has read the data item b written by $W_0(b)$. Then, at every node, order of execution should be W_0 followed by W_2.

4. Monotonic Writes: A write is executed at a node if it includes all previous writes that were executed in the session.

   Example 2.4: Let $W_1(b)$ occur before $W_2(c)$ in the same session. Then, at any node, when W_2 is being executed, W_1 would have already been executed.

Providing session guarantee in a transactional framework becomes more complex than that considered in [TDPSTW94], as a transaction usually consists of multiple operations. While ensuring session guarantee for a transaction, only the previous transactions (both active and inactive transactions) in the session which directly or indirectly conflict with the transaction are considered. That is, if the present transaction does not conflict (directly or indirectly) with any of the previous (active and inactive) transactions in the session, then the present transaction need not obey any of these session guarantee types. We provide our own classification of session guarantee where we consider read only transactions (ROTs) and update transactions, separately. Types of session guarantees are ordered based on their strength; the latter ones being stronger than the former.

1. Monotonic Updates: An update transaction is executed at a node, if all the preceding conflicting update transactions executed in the system in the same session have been executed already at that node. This session guarantee is the weakest of all types of session guarantees. This is similar to Monotonic Writes guarantees in [TDPSTW94].

2. Updates Follow Read Only: At every node, an update transaction follows those update transactions whose effects were seen by the ROTs in the same session. This session guarantee is similar to Write Follows Reads guarantees in [TDPSTW94]. It is stronger than Monotonic Update guarantees as ordering between a ROT and an update transaction is considered stronger than ordering between two update transactions. Please note that Monotonic Update guarantees and Update Follows Read Only guarantees are essential to ensure 1SR in a replicated system.
3. Read Only Follow Updates: At a node, a ROT sees the effects of all the update transactions (i.e., conflicting transactions as the ROT can only see preceding conflicting transactions) being executed in the same session. This is similar to Read Your Writes guarantees in [TDPSTW94]. This type of session guarantee is stronger than the above two, as without this criterion 1SR may be ensured.
4. Monotonic Read Only: At a node, a ROT sees the effect of all previous transactions being executed by the same session. That is, a ROT sees the effect of both the ROTs and update transactions executed previously. This is similar to Monotonic Reads in [TDPSTW94]. This is the strongest of all the above types of session guarantees, as even two ROTs are ordered globally. As both transactions of the session are ROTs, this criterion is not required to ensure 1SR.

Chapter 3

Replication Model

This chapter introduces the basics of the replication model on which our protocol is based. First, we start with the architecture where we describe the configuration used in UDDI. Then, we describe the communication model used by our protocol. Later, we explain a transactional model with a session guarantee mechanism.

3.1 Architecture

We consider a fully replicated registry as adopted in Version 3 of UDDI specification [UDDI]. In this system, each data item is replicated at all the nodes. Our model facilitates complete distribution and loose synchronization among the nodes. In a distributed registry with n nodes, each data item is in the *custody* of exactly one node. In UDDI, changes to a data item must be first executed at the custodian node and these changes are propagated and executed at other nodes. Different data items may be in custody of different nodes. Therefore, changes to a set of data items may be executed at different nodes and all the latest updates can be found at no single node. In our model, the primary transaction executes the transaction at a node in the registry. Later, changes of the transaction are propagated as a change-record to other nodes. Each change-record is associated with a monotonically increasing unique number, called the Unique Sequence Number (USN), which is assigned at the node where the primary transaction is executed. It should be noted that the USN generated

at a node should not be compared with USNs generated at another node (i.e., USN is not globally unique). Change-records are implemented as refresh transactions. Our protocol minimizes communication, coordination, and synchronization required for execution of primary transactions. We assume the fail stop model where a faulty node stops functioning completely [SS83].

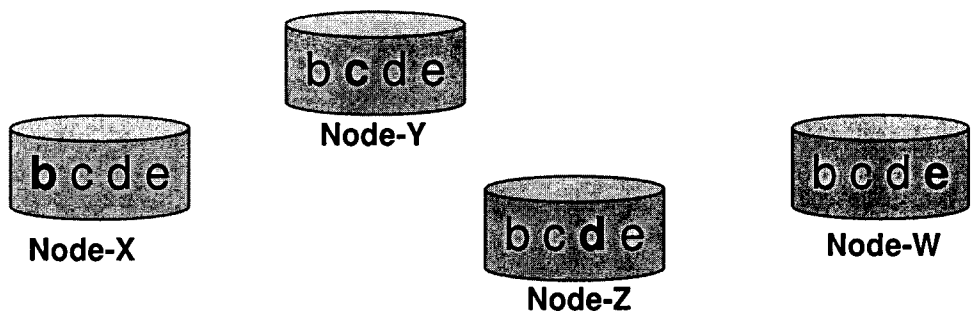

Figure 3.1 Illustrates the configuration in UDDI

Figure 3.1 shows the configuration of UDDI. Node-X is the custodian of data item b. Similarly, data items c, d, and e are in custody of Node-Y, Node-Z, and Node-W, respectively. A change to data items b, c, d, and e is first updated at Node-X, Node-Y, Node-Z, and Node-W, respectively.

3.2 Communication Model

We assume that the system provides a *reliable broadcast* of the messages. Reliable broadcast ensures that messages sent by a correctly working node are received by all the correctly working nodes eventually in the same order. The reliable broadcast of messages does not impose any ordering on messages at the global level. If Node-X sends message Υ and Node-Y sends message ϕ, these two messages may be delivered in different orders at different nodes. Other types of broadcast primitives are *causal* and *total order*. The total order broadcast ensures that all messages are delivered in the same order at all the nodes. The causal broadcast ensures that if the broadcast of message ϕ causally precedes the broadcast of message Υ, then at no node, is Υ delivered before ϕ.

The reliable broadcast provides the weakest ordering guarantee of the three types of broadcast primitives. Total ordering imposes a total order on all the messages in the system. This requirement is too strong in a distributed transaction processing environment, such as replication. It causes reduced concurrency, resulting in lower transaction throughput of the system. A better tradeoff between these two extremes is the causal broadcast, which imposes partial ordering of messages such that any two messages are ordered at the global level only if there is a happens-before relationship between them. When this is translated to the transaction processing context, it means that two transactions are ordered if the change-record message of the second transaction is sent after that of the first transaction has been received at a node. This is

termed as the *false causality* in [TG98] where two messages are ordered just because the second message occurs after the first, but the first has not caused the second event to occur. In the transaction processing context, we avoid the false causality among the messages, by inducing ordering between two messages only if there is a dependency between transactions.

Example 3.1: Consider the setup as shown in Figure 3.2. We illustrate the false causality using the following transactions:

$T_K = W(b)$

$T_L = W(c)$

$T_H = R(b)\ W(c)$

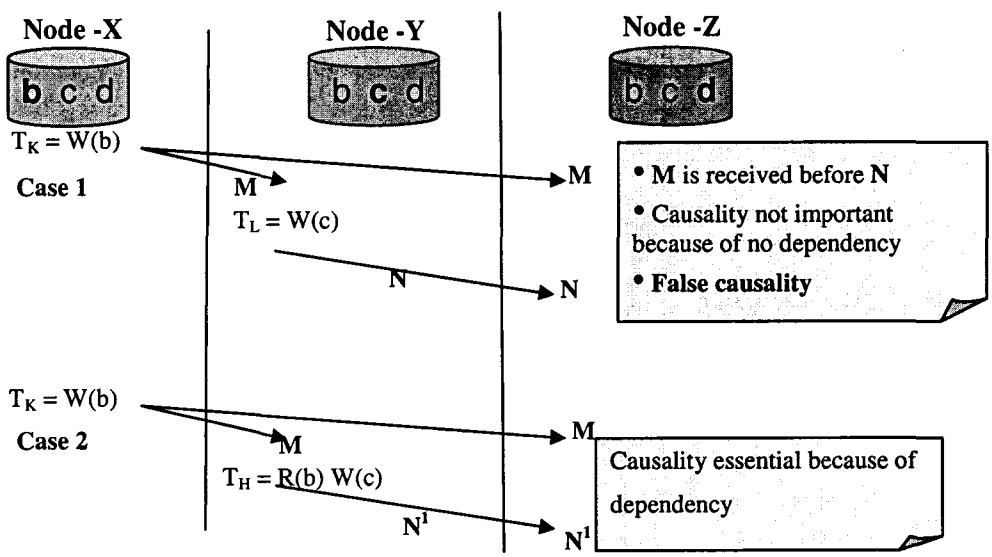

Figure 3.2 Illustrates the false causality

In case 1, Node-X executes T_K and sends M as the changes of T_K to Node-Y and Node-Z. Node-Y after receiving M executes T_L and sends N. This happens-before relationship at Node-Y orders M before N at all the nodes. This ordering is not important, as T_K and T_L are independent transactions. This communication primitive is the false causality. In case 2, Node-X sends M to Node-Y and Node-Z. Node-Y receives M and executes T_H and sends N^1. Because of the dependency between T_K and T_H, M before N^1 ordering has to be maintained at all the nodes. Therefore, causality is important in this case.

Facilitating the flexibility for delivery of independent transactional messages in any order at all the nodes yields higher concurrency in distributed transaction processing.

3.3 Transactional Model

A transaction basically consists of a set of read and write operations. A transaction is usually associated with ACID properties:

- A - Atomicity: Either all operations of a transaction are executed or none of them is executed.
- C - Consistency: Guarantees that the execution of the transaction transforms the database from one valid state to another.
- I - Isolation: Noninterference of concurrent transactions. That is, one transaction will not see the intermediate values of the other.

- D - Durability: Committed updates are never lost, and their effect persists in the database beyond the transaction's lifetime.

A TM and scheduler of the database system take care of preprocessing, scheduling, and execution of all the transactions in the system. For simplicity, in the rest of the thesis, we abstract that the TM will take care of all these functionalities. Basically, the concurrency control mechanism employed for execution of transactions can be classified as *pessimistic* or *optimistic*. Typically, locking is employed for the pessimistic concurrency control where locks are first acquired on all the data items and then the transaction is executed and committed. In the optimistic method, the transaction is first executed and then it is validated to find out if the execution is correct. Only after the successful validation, does the transaction commit. The *Two phase locking* (2PL) is a type of locking mechanism where all the locks required by the transaction are acquired in the first phase. In the second phase, those locks can be released gradually. It should be noted that once 2PL enters the second phase, no locks can be acquired further. This clearly distinguishes a growing phase which is followed by a shrinking phase. If a scheduler acquires all the required locks before the execution starts, then it is called a *conservative scheduler*. Otherwise, it is called an *aggressive scheduler*. If all the locks are released atomically in the shrinking phase, the locking mechanism is called the *Strict Two Phase Locking* (strict-2PL).

Two transactions are said to be conflicting if they access the same data item. Based on the type of operations accessing the data item, conflicts are classified into following types:

- WW conflict: The operations of both transactions are write operations.
- WR conflict: The operation of the first transaction is a write and that of the second transaction is a read.
- RW conflict: The operation of the first transaction is a read and that of the second transaction is a write.
- RR conflict: The operations of both transactions are read operations.

We have listed RR conflicts for the sake of completeness. Usually, and in the rest of the thesis, a conflict refers to either of the WW, WR or RW conflicts.

In distributed transaction processing, execution of a transaction involves participation of a number of nodes. In order to ensure ACID properties of transactions, atomic commitment protocols, such as the *Two Phase Commit* (2PC), are used. 2PC has two phases — voting phase and decision phase. In this protocol, one of the nodes is selected as a coordinator. In the voting phase, the coordinator sends a vote request to all the other nodes known as participants. Each of the participants sends Yes or No vote to the coordinator. In the second phase, the coordinator, based on the responses received, decides either to commit or abort the transaction. If all the participants vote Yes, the protocol decides to commit. Else, if any of them vote No, it decides to abort.

This decision message is sent to all the participants. Upon receiving the decision message, based on the decision, each of the participants either commits or aborts the transaction. Hence all the nodes reach the same consistent state.

A history (H) indicates the order in which the operations of the transactions are executed relative to each other. The serialization graph (SG) for H, is a directed graph whose nodes are the transactions that are committed in H and whose edges are all $T_K \rightarrow T_L$ ($K \neq L$) such that one of T_K's operations precedes and conflicts with one of T_L's operations in H.

In replication, ensuring atomicity of execution of a transaction at all the nodes is not a feasible option. Therefore, among commercially available databases, a relaxed atomicity criterion, such as lazy replication, is popular. In this thesis, we consider a lazy replicated database which uses the locking mechanism. We assume that each node has its own TM and uses the locking mechanism, such as 2PL, with an added flexibility of giving locks to transactions being executed at other nodes. In short, one node can request a lock from another node. In our system, we assume that the custodian node of a data item acts as a lock granting agency for that particular data item.

3.3.1 Local and global transactions

We classify transactions as local and global based on communication and coordination requirement for the execution of the primary transaction. This

classification is based purely on the execution location of the transaction. The execution location is decided by types of operations and data items the transaction accesses. The difference between local and global transactions is that a local transaction does not require any communication or coordination with any other nodes in the system for a primary transaction to commit, whereas a global transaction does.

Example 3.2: Consider the setup as shown in Figure 3.3. Node-X, Node-Y, Node-Z, and Node-W are custodians of data items b, c, d, and e, respectively. Following are the set of transactions executed in the system.

T_K = W(c)

T_L = R(b) R(c) R(d)

T_H = W(b) W(e)

T_G = R(b) W(d)

Transactions are classified into the following categories and sub-categories:

1. Local Transactions:

    1.1. ROTs: As we consider the fully replicated registry, a ROT can read any data item at a node. In Figure 3.3, T_L is a local transaction, as it is a ROT.

    1.2. Local update transactions: These transactions access data items in the custody of a single node and are executed at its custodian node. In Figure 3.3, T_K is a local transaction.

2. Global Transactions: A primary update transaction accessing data items in the custody of remote nodes is a global transaction.

2.1 Read access at other nodes: The coordinator of a primary transaction accesses data items in custody of other nodes in read operations. In Figure 3.3, T_G is a global transaction.

2.2 Write access at other nodes: The coordinator of a primary transaction accesses data items in the custody of other nodes in write operations. In Figure 3.3, T_H is a global transaction.

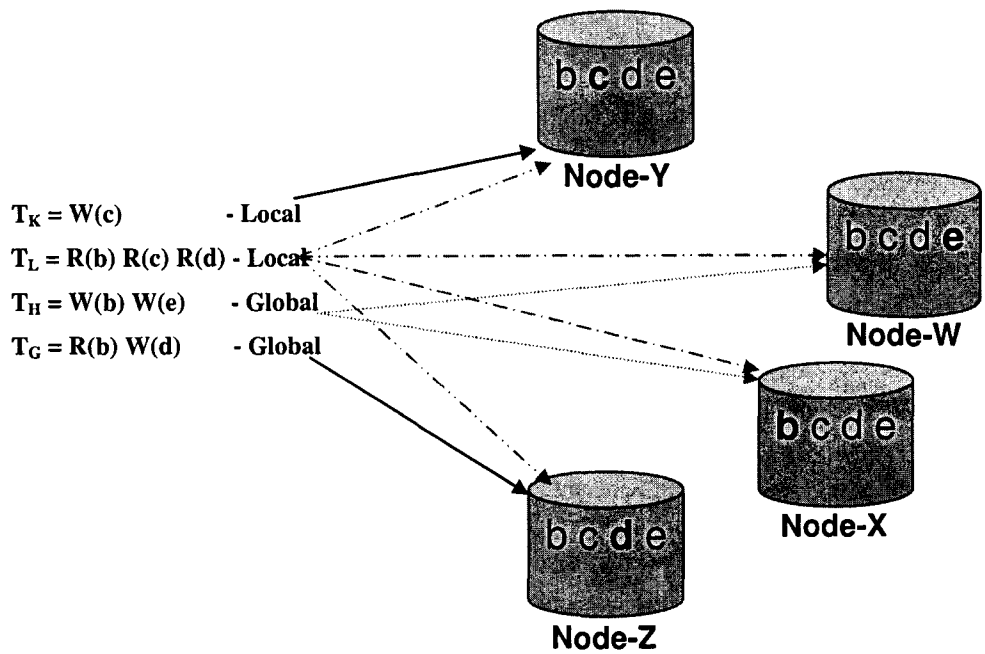

Figure 3.3 Illustrates of global and local transactions

In order to execute the primary transaction of a global transaction, we differentiate the roles of nodes in the system into the following types:

1. Coordinator Node: Usually, one of the custodians of write data items is selected by the system as the coordinator. Even a node with the custody of read data item or which is not in custody of any data item of a transaction may become coordinator, but it would increase communication costs and make the data item inaccessible to other concurrent primary transactions.
2. Participant Node: Custodians of other read and write data items of a global transaction are termed as participant nodes.
3. Non-Participant Node: All the remaining nodes are termed as Non-Participants.

	Type	**Coordinator**	**Participants**	**Non-Participants**
T_K = W(c)	Local	Node-Y	-	All others
T_L = R(b) R(c) R(d)	Local	Any node	-	All others
T_H = W(b) W(e)	Global	Node-W/X	Node-X/W	Node-Y & Z
T_G = R(b) W(d)	Global	Node-Z	Node-X	Node-Y & W

Table 1 Classification of transactions

Let us consider the assignment of these roles for transactions T_K, T_L, T_H, and T_G.

1. **T_K**: Usually, T_K is executed at the coordinator node, Node-Y, as it is the custodian of data item c.

2. **T_L**: It is a local transaction which can be executed at any node.

3. **T_H**: Usually, the system will select either Node-W or Node-X as a coordinator for execution of T_H, as the custodians of the write data items are Node-W and Node-X. If Node-W is the coordinator, Node-X will be the participant node or vice versa. Node-Y and Node-Z are non-participant nodes.

4. **T_G**: Usually, T_G is executed at Node-Z as it is the custodian of the write data item of the transaction. Node-X is a participant node. Node-Y and Node-W are non-participant nodes.

3.4 Session Guarantee

In this thesis, we facilitate a user in a session to observe a registry with a view that is increasingly up-to-date over time. This can be achieved by ensuring Read Only Follow Updates and Monotonic Read Only guarantees. We know that all the nodes in the registry start with the same initial state. Any new transaction executed in the system increases the state of the system. The primary transactions can be executed at different nodes in the system. A state transition diagram in our model is given in Figure 3.4.

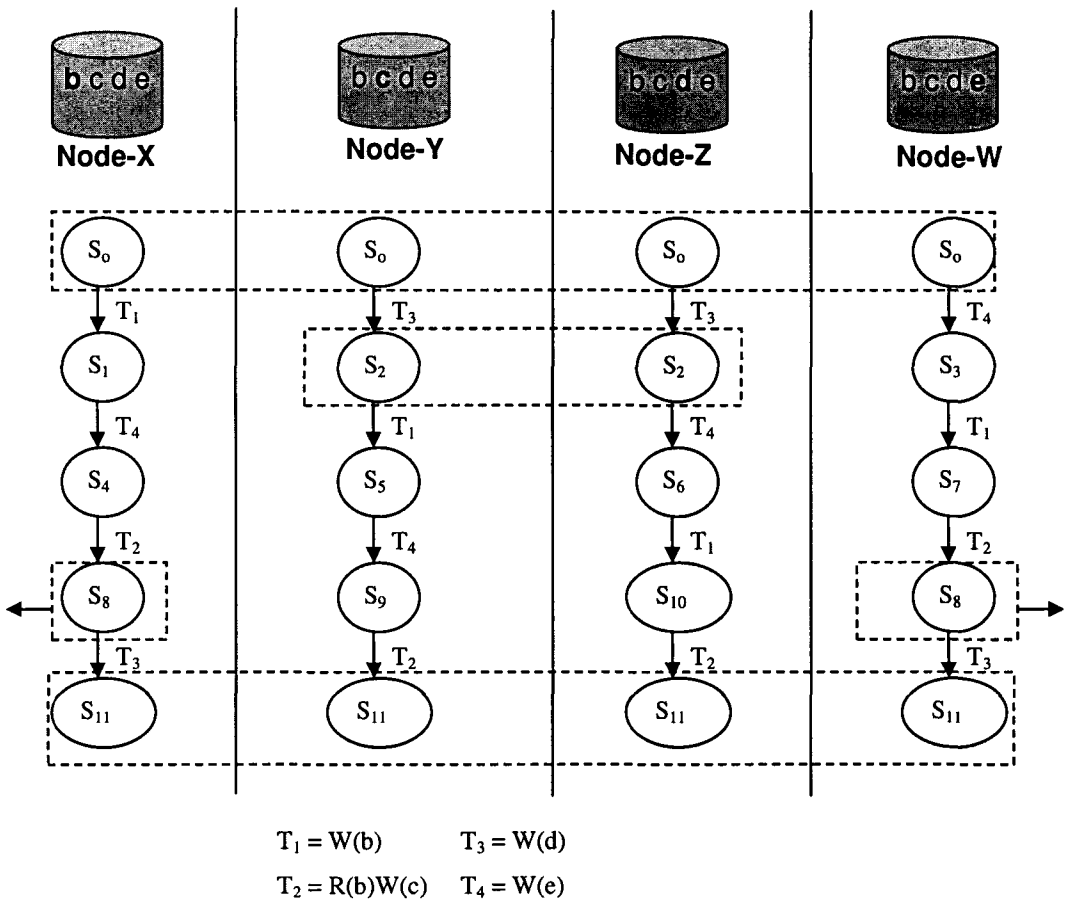

Figure 3.4 Illustrates the state transitions among the nodes in the registry

Example 3.3: Consider the setup shown in Figure 3.4. S_0, S_1, S_2,...,S_{11} indicate the states of nodes in the system. The global view of the registry indicating the present states of all the nodes cannot be obtained anywhere. Just for the sake of clarity, we

explain with a global view which would have occurred in the system. Execution of a transaction triggers a transition from one state to another.

Let us consider the execution of T_1 at Node-X. T_1 triggers a transition from state S_0 to S_1. The transactions in the system are T_1, T_2, T_3, and T_4. There is no unique sequence in the execution of all these transactions. That is, the sequence of execution of transactions is different at different nodes. All the nodes in the registry start with the same initial state S_0 and end with the same final state S_{11}. There is no intermediate shared state between all these nodes. There is an intermediate state shared by Node-Y and Node-Z, namely S_2. Also, S_8 is shared by Node-X and Node-W.

Example 3.4: Consider the setup shown in Figure 3.5. T_{11} and T_{12} are executed by User-A in a session. T_{21} and T_{22} are executed by User-B in another session. User-A's transactions are as follows:

T_{11} = W(b)

T_{12} = R(b) W(c)

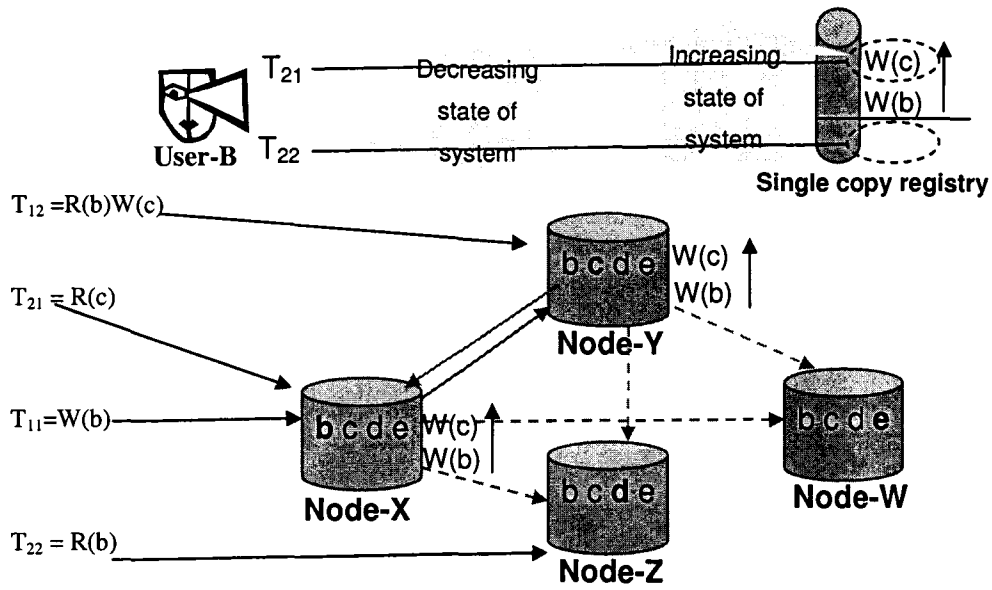

Figure 3.5 Snapshot of UDDI registry to illustrate session guarantee

The primary transactions of T_{11} and T_{12} are executed at Node-X and Node-Y, respectively. As we employ a lazy replication scheme, we obtain a snapshot as shown in Figure 3.5, after execution of the following set of events in the given order.

a. The primary transaction of T_{11} is executed at Node-X and the change-record is sent to all other nodes.

b. The change-record of T_{11} is received and executed at Node-Y.

c. The primary transaction of T_{12} is executed at Node-Y and the change-record is sent to all other nodes.

d. The change-record of T_{12} is received and executed at Node-X.

e. Change-records of neither T_{11} nor T_{12} have been received at Node-Z or Node-W

User-B's transactions in another session are as follows:

$T_{21} = R(c)$

$T_{22} = R(b)$

As the registry is fully replicated, these two transactions can be executed at any node by the system. The user will not be aware about where transactions are executed because of replica transparency. At this point in time (referred to in Figure 3.5), the following transactions are executed in the given order:

a. T_{21} is executed at Node-X

b. T_{22} is executed at Node-Z.

User-B obtains the latest value of the registry when T_{21} is executed at Node-X. When he executes T_{22} at Node-Z, neither T_{11} nor T_{12} have been executed at that node so far. Therefore, he obtains the initial value of data item b in the registry.

This is an inconsistent view because when one copy equivalence is considered, User-B who has seen $T_{12,}$ has not been able to see the previous conflicting transaction (T_{11}). As shown in Figure 3.5, User-B is said to have seen the decreasing state of the registry. This problem arises basically due to the lazy replication in a replica

transparent environment. This can be solved using session guarantees. In [DS05], this is termed as *transaction inversion* where a ROT precedes an update transaction in the serialization order, despite the fact that it follows the update transaction in the client's request stream.

In general, as different transactions may execute the primary transaction at different nodes, the sequence of states at all the nodes may not be the same. If a user always reads and updates at only one node, the session guarantee is ensured trivially. This may be a major restriction, as efficient load balancing cannot be achieved. We provide the flexibility for consecutive transactions in a session to access different nodes.

Our basic protocol explained in chapter 4 ensures 1SR. In order to ensure 1SR, it considers only update transactions in the system. This is because ROTs executed at different nodes may cause indirect conflicts and create a cycle in the global SG. Our protocol executes all update transactions such that, it constructs an acyclic SG at the global level. The ROT can read some consistent state from this global SG. The session guarantee mechanism built on basic protocol ensures that the ROTs see the increasing state of the system.

Chapter 4

Replication Protocol

The replication of UDDI poses new challenges as compared to conventional replicated databases. This is, basically, because of the concept of custodianship employed in UDDI. There are many configurations employed for lazy replication in the literature. [HSAE03] considers a replicated database, where every copy is a master copy. In this configuration, primary transactions can be executed at any node. [HSAE03] uses an optimistic approach, which leads to an increase in the abort rate with increase in the number of nodes and subsequent poor performance as the conflict rate increases. [DS04] considers the primary master-slave approach. With respect to the execution of primary transactions, this is a centralized approach, where all the primary transactions are executed at the same node. This configuration does not facilitate efficient load balancing as all the transactions have to be executed at the same node. This leads to poor performance, when the cost of execution (in terms of the time required to execute) is higher than the cost of communication (in terms of the time required to communicate). Also, the failure of the primary node causes loss of all the latest updates in the system. With respect to execution of the primary transaction, configuration in UDDI is different from the approaches studied in literature described above. In this chapter, we design a protocol which suits the above discussed requirements and is efficient for replicated UDDI registries.

4.1 Protocol

Our protocol uses a locking mechanism. The locking mechanism is an extension of 2PL which is designed such that once the primary transaction executes, the transaction's serialization order is fixed in the acyclic global serialization graph. Each node in the registry has its own TM. Usually, a TM uses 2PL [BHG87] for the execution of refresh transactions and ROTs. A node in the registry can request locks from another node for the execution of the primary transaction.

Let us consider a registry with n nodes. Each transaction is associated with an Update Sequence Number (USN), which is unique at the node where its primary transaction is executed. The USN at a node increases monotonically. A transaction in a registry is uniquely identified by the pair <Node-ID, USN>. A state of the node can be defined as transactions that have been committed successfully at that node. Each of the nodes in the registry has a set of data structures. We list them with respect to a node, say Node-Y:

1. A two dimensional array $N_Y[1,2,3,...,n][1,2,3,...,n]$ (denoted by N-array) indicates the state of Node-Y. $N_Y[y][y]$ is the USN of the latest primary transaction, T_K, that was executed at Node-Y and has been committed successfully at that node. $N_Y[x][z]$ indicates the USN of the latest transaction whose primary transaction was executed at Node-Z with Node-Z as its coordinator and has been executed at Node-X that Node-Y is aware of. Thus, $N_Y[x][1,2,3,...n]$ denotes the USNs of the

set of the latest transactions that have been executed at Node-X whose primary transactions were executed at Nodes 1,2,3,...,n, that Node-Y is aware of.

2. A one dimensional present state array $P_Y[1,2,3,...,n]$ (denoted by P-array) is $N_Y[y][1,2,3,....n]$. $P_Y[x]$ denotes the USN of transaction whose primary transaction was executed at Nodes-X and also has been executed at Node-Y.

3. A one dimensional array $M_Y[1,2,3,...,n]$ (denoted by M-array) indicates the set of the latest transactions which have been executed at all the Nodes 1,2,3,...,n that Node-Y is aware of. That is, $M_Y[x]$ denotes the USN of the latest transaction whose primary transaction was executed at Node-X and has been executed at all the other nodes, which Node-Y is aware of.

After transaction T_K is executed at the coordinator node, updates of the transaction are broadcast as a change-record to the other nodes. The USN of a transaction is assigned (at commit time of the primary transaction) to a change-record. A change-record contains the following fields:

1. Node identifier of the coordinator of T_K;

2. The USN of transaction T_K;

3. Write operations with its values and read operations of transaction T_K;

4. The present state array ($P_Y[1,2,3,...,n]$) of the coordinator (optional); and

5. The dependency array (D-array) of T_K. $D_K[1,2,3,...,n][1,2,3,...,n]$ of T_K contains the USN of T_K and USNs of all the preceding conflicting (directly and indirectly) transactions.

A Change-record is stored at a node until it is explicitly deleted by the *delete_change-record* procedure.

Our protocol implements causal broadcast of messages. It ensures causal delivery of messages only among the messages of conflicting transactions, and hence, avoids the issue of false causality [TG98]. Messages which have to be delivered causally are change-record, lock-grant, and acknowledge messages. We provide separate protocols for each type of transactions — primary, refresh, and read only. Primary transactions request PT locks (PT-S and PT-X, representing read and write locks, respectively). PT locks are used to resolve the conflicts and decide the transaction's serialization order. Refresh transactions request RT locks. Our protocol is such that when a transaction has been granted a PT lock on a data item, its conflicting transaction can not be granted a RT lock. The local TM executes ROTs and refresh transactions (both local transactions) such that they ensure conflict serializability. All three types of protocols can be executed independently.

4.1.1 For primary transaction T_K

The primary transaction protocol is basically a 2PL. Execution of a primary transaction has mainly five stages:

(1) Lock acquisition phase

a) Requests locks on data items which are in custody of the local node, atomically.

b) May request a few locks at remote nodes also (for a global transaction).

(2) Execution phase (executes the transaction)

(3) Validation phase

a) Requests locks at remote nodes from which they were not requested earlier (for a global transaction).

(4) Abort phase (aborts & releases locks)

a) Sends lock-release-request to nodes from which locks were requested earlier.

(5) Commit phase (executes refresh transaction protocol, commits, creates change-record, releases locks and broadcasts change-records).

When a transaction starts, the coordinator node acquires locks on all the data items of which it is the custodian in stage (1.a), atomically. For a global transaction, the coordinator acquires locks for data items in custody of other nodes at their respective custodian nodes, either in stage (1.b) or (3.a). If all the required locks are obtained in stage (1), then execution is pessimistic. Otherwise, it is optimistic. The protocol executes either stage (4) or stage (5). We do not consider deadlock initially. We assume that when the TM at one node requests a lock from some other node, it will

obtain the lock within a finite amount of time. Figure 4.1 illustrates the primary transaction execution using a flow chart. The flow chart shows that if the primary transaction is executed pessimistically, then the validation phase is not executed. It executes either stage (4) or stage (5).

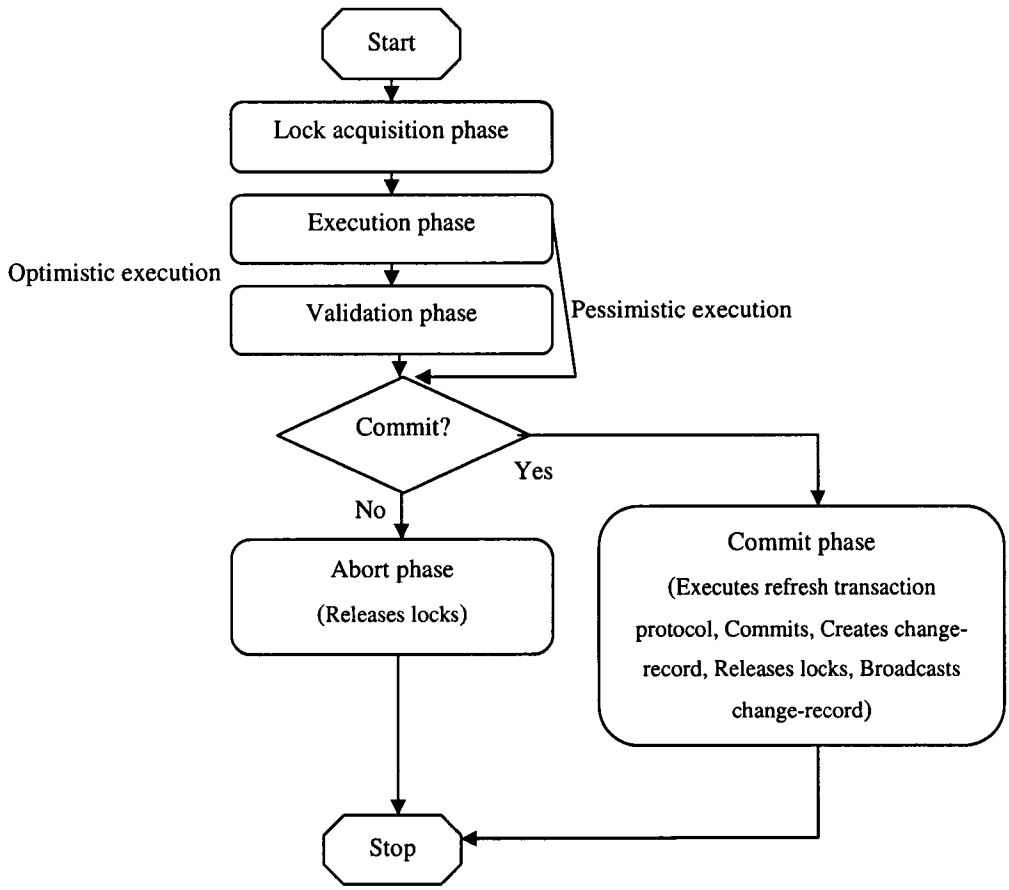

Figure 4.1 Flow chart representing the execution of the primary transaction protocol

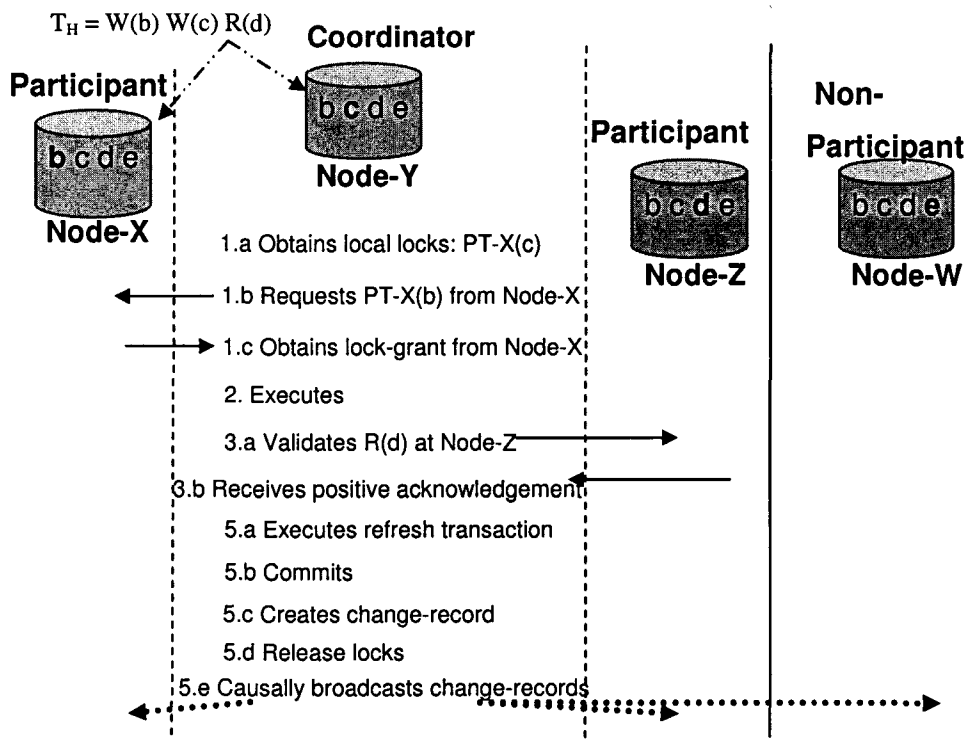

Figure 4.2 The primary transaction protocol with the coordinator's view for the execution of T_H

Example 4.1: Consider the setup shown in Figure 4.2. Node-X, Node-Y, Node-Z, and Node-W are custodians of data items b, c, d, and e, respectively. Let us consider the execution of the transaction T_H, which writes b, c, and reads d. (Please note that the T_H is different from T_K used in the description of the protocol as T_H in the example illustrates only limited set of cases.) The write data items b and c are in the custody of Node-X and Node-Y, respectively. The system selects Node-Y as the coordinator to execute T_H. Node-Y requests the PT-X lock on data item b from Node-X,

pessimistically. The protocol reads the data item d, which is in custody of Node-Z, optimistically. The coordinator validates execution of the read operation of the transaction from Node-Z in stage (3). Participants are Node-X and Node-Z. Non-Participant is Node-W.

The execution of the primary transaction can be explained from the coordinator's and the participant's view.

For Coordinator (Coordinator's view)

Consider the execution of the primary transaction, T_K, at the coordinator, Node-Y.

(1) Lock acquisition phase: T_K acquires the locks in two steps, namely, stage (1.a) and stage (1.b), executed in the given order. Once all the requested locks have been acquired the protocol goes to stage (2).

   (a) Requests PT-S and PT-X locks on the data items to which Node-Y is custodian, atomically. That is, T_K obtains locks on all the data items in custody of Node-Y or does not obtain any lock. Only after completing stage (1.a) the protocol goes to stage (1.b).

   (b) The coordinator of T_K decides which locks to request from participants now (stage (1.b)). It sends the PT-X/PT-S lock-request message (contains the type of locks and the data items on which locks are required) to the corresponding custodian nodes.

(2) Execution phase: Transaction T_K is executed. All writes are performed in a private workspace.

(3) Validation phase: T_K validates the operations for the data items from which locks were not requested in stage (1). Validation is performed by sending messages to the corresponding participants using one or more of the following types of messages:

(a) Read validate a data item: If a read operation on data item d is to be validated, then S-optimistic-request on data item d is sent to the custodian of data item d. The message contains transaction identifier, <Node-ID, USN>, from which data item d was read. (This can be obtained from the change-records stored locally.)

(b) Write validate a data item: If a write operation on data item b is to be validated, then X-optimistic-request on data item b is sent to the custodian of data item b.

(c) Read and write validate a data item: If the transaction has both read and written data item e without requesting locks pessimistically, then both X-optimistic-request and S-optimistic-request on data item e are sent to the custodian of data item e.

If the coordinator obtains a negative response from at least one participant, then the protocol goes to stage (4). Otherwise, upon arrival of positive responses from all its participants, it goes to stage (5).

(4) Abort phase: The protocol aborts transaction T_K, releases all its PT locks (i.e., PT-X and PT-S) at the local node, atomically. Then, sends a lock-release message to all the other participant nodes.

(5) Commit phase: The protocol performs each of the following steps in the given order:

a) Executes the refresh transaction protocol (explained later as the refresh transaction) for T_K.

b) Commits the primary transaction, T_K.

c) Generates the USN. Then, updates the state array, N_Y, and creates the present state array, P_Y, the D-array, D_K. Creates change-record for T_K.

d) Releases all the PT locks.

e) Broadcasts change-records to all the nodes.

Then, *session_update* procedure is executed. This procedure is discussed in section 4.1.4.

The data-structures at all the nodes in the registry have to be initialized when the replicated registry starts. Initially, there are no transactions in the system. Therefore, the state arrays at Node-X, Node-Y, Node-Z, and Node-W are initialized to zero value as follows:

$N_X[1,2,3,...,n][1,2,3,...,n]:= N_Y[1,2,3,...,n][1,2,3,...,n] := N_Z[1,2,3,...,n][1,2,3,...,n] := N_W[1,2,3,...,n][1,2,3,...,n] := 0$

During the execution of the primary transaction of T_K, arrays associated with the change-records are updated or created as follows:

- Update of the N-array, $N_Y[1,2,3,...,n][1,2,3,...,n]$ at Node-Y

$N_Y[y][y]$:= USN of primary transaction T_K

- Creation of present state array (P-array) P_Y representing y^{th} row of N_Y

$P_Y[1,2,3,\ldots,n]$:= $N_Y[y][1,2,3,\ldots,n]$

- Creation of D-array D_K for T_K

$D_K[1,2,3,...,n][1,2,3,...,n] := 0$

$D_K[y][y]$:= USN of transaction T_K

For all x, which are participants of T_K, $D_K[x][y]$:= USN of transaction T_K

Let transaction T_K conflict with a set of transactions namely T_{K1}, $T_{K2},\ldots,T_{KP}$. WW, RW, and WR conflicts with T_K are calculated at the local TM as follows:

The change-records are stored at a node in the order of their execution. That is, the change-record of the latest transaction is stored at the top of the queue and that of the oldest transaction at the bottom. For each of the operations of T_K, the conflicting operation in the change-record is searched, starting from top and then proceeding towards the bottom of the queue at that node. If a conflicting operation in the change-record of T_L is found, then T_L conflicts with T_K. The types of conflicts between T_K and T_L can be WW, WR, and RW. This procedure is continued for all the remaining

operations of T_K. For a global transaction, instead of searching at a local node for the conflicting transaction (i.e., searching in change-records using the above procedure) on a data item in custody of another node, D_{KRM}-array received along with lock-grant/acknowledgement messages can be used. While the former method would also give the same result, the latter eliminates re-computation costs.

The computation for D_K of transaction T_K can be performed as follows. Let D-arrays in change-records of conflicting transactions found at local nodes, namely T_{K1}, T_{K2},..., T_{KM} be D_{K1}, D_{K2},..., D_{KM}, respectively. Let D-arrays in the lock-grant/positive-acknowledgement messages of T_{KM1}, T_{KM2},..., T_{KP} be D_{KM1}, D_{KM2},..., D_{KP}, respectively. Then, the code finds the most recent transactions that were executed at each node in the registry that conflict with T_K as follows:

for var = 1 to n, where n is the number of nodes
  for varp = 1 to n, where n is the number of nodes
    D_K[var][varp] := Max(D_{K1}[var][varp], D_{K2}[var][varp],...,D_{KM}[var][varp], D_{KM1}[var][varp], D_{KM2}[var][varp],...,D_{KP}[var][varp],..., D_K[var][varp])
    Function Max returns the maximum integer for a given set of integers (throughout the thesis, this function is used with the same meaning).

The above code computes and stores in D_K[1,2,3,...,n][1,2,3,...,n], USNs of all the transactions conflicting with T_K, including itself. Therefore, D_K[x][y] contains the USN of the transaction conflicting with T_K, or the USN of T_K, which was executed with Node-Y as the coordinator and Node-X as the participant.

In Example 4.1 (refer Figure 4.2), the coordinator Node-Y, requests a PT-X lock from Node-X in stage (1). After the coordinator obtains the lock on data item a, primary

transaction T_H is executed in stage (2). In stage (3), the transaction sends the validation request message to Node-Z. Node-Z validates and obtains the PT-S lock on data item c and sends a positive acknowledgement to Node-Y. Then, the transaction commits in stage (5).

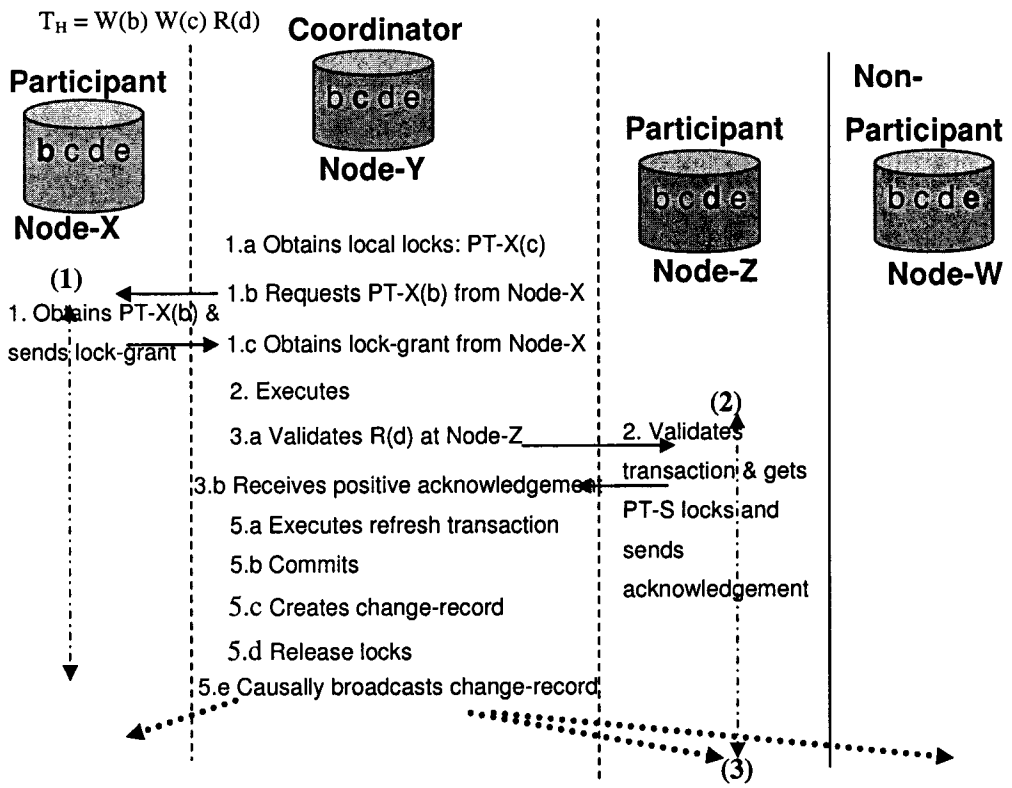

Figure 4.3 The primary transaction protocol with participant's view for T_H

For Participants (participant's view):

Consider the execution of the primary transaction, T_K, at the participant, Node-X.

(1) Upon receiving a PT-S/PT-X lock-request (refer to stage (1.b) in the coordinator's view): Node-X acquires the corresponding PT locks (PT-S or PT-X) from the local TM. Then, the local TM sends the lock-grant message which contains D-array $D_{KMI}[1,2,3,...,n][1,2,3,...,n]$ (creation of D_{KMI} is explained later).

(2) Upon receiving a validation request message (refer to stage (3) in the coordinator's view): Node-X can receive S-optimistic-request or X-optimistic-request or both on data item d. Validation of the operation on a data item is performed as follows:

a) S-optimistic-request: On receiving an S-optimistic-request on data item d, a check is performed if there are any recent writes on data item d. This is performed by searching the change-records stored at the local node for write on data item d, starting from the latest change-record until either the change-record with identifier, <Node-ID, USN>, (identifier of the change-record from which the data item has read from at the coordinator) or the last change-record is reached. If any change-record with such an operation is found, then a negative acknowledgement is sent. Otherwise, the PT-S lock is obtained from the local TM (we assume that checking the condition and obtaining the locks are performed atomically) and a positive acknowledgement is sent.

b) X-optimistic-request: On receiving an X-optimistic-request on data item d, Node-X obtains the PT-X lock from the local TM and sends a positive acknowledgment.

c) Both S-optimistic-request and X-optimistic-request: On receiving both of these requests on data item d, a check is performed as in the case of S-optimistic-request on data item d. If successful, PT-X lock on data item c is acquired and a positive acknowledgement is sent.

All the types of acknowledgement messages contain D-array, $D_{KMI}[1,2,3,...,n][1,2,3,...,n]$ for i^{th} request of T_K.

(3) Commit phase: The refresh transaction protocol is executed upon receiving a change-record and then all the PT locks are released.

(4) Upon receiving the lock-release: Node releases all the PT locks held by the transaction, T_K.

Noted that in the above protocol, the primary transaction can abort due to S-optimistic-request but it will never abort due to X-optimistic-request.

Creation of D-array which is associated with lock-grant/acknowledgement messages for transaction T_K is performed by the TM at Node-X, similar to the procedure explained in the coordinator's view. The only difference is that, in the coordinator's view, the conflicts are detected on all the operations of T_K; whereas, in the participant's view, the conflicts are detected only on the requested data item.

In Example 4.1 (refer Figure 4.3), Node-X receives a lock-request message from Node-Y on data item b. After the TM at Node-X obtains the lock, it sends the lock-grant message to Node-Y. Node-Z receives an S-optimistic-request message on the

data item d from Node-Y. It validates the read operation of the transaction. Then, it obtains the PT-S lock and sends a positive acknowledgement message.

4.1.2 For refresh transaction T_K

Usually, the locking mechanism is used for the execution of the refresh transaction protocol. The execution of refresh transaction protocol in each of the roles can be explained as follows:

- Coordinator node: The refresh protocol is nested inside stage (5) of the primary transaction of the coordinator's view.
- Participant node: The refresh protocol is nested inside stage (3) of the primary transaction of the participant's view.
- Non-Participant node: The refresh protocol is executed independently.

The protocol has three stages:

1. Lock acquisition phase
2. Execution phase
3. Commit phase

The protocol is the same for the coordinator, participant and non-participant nodes.

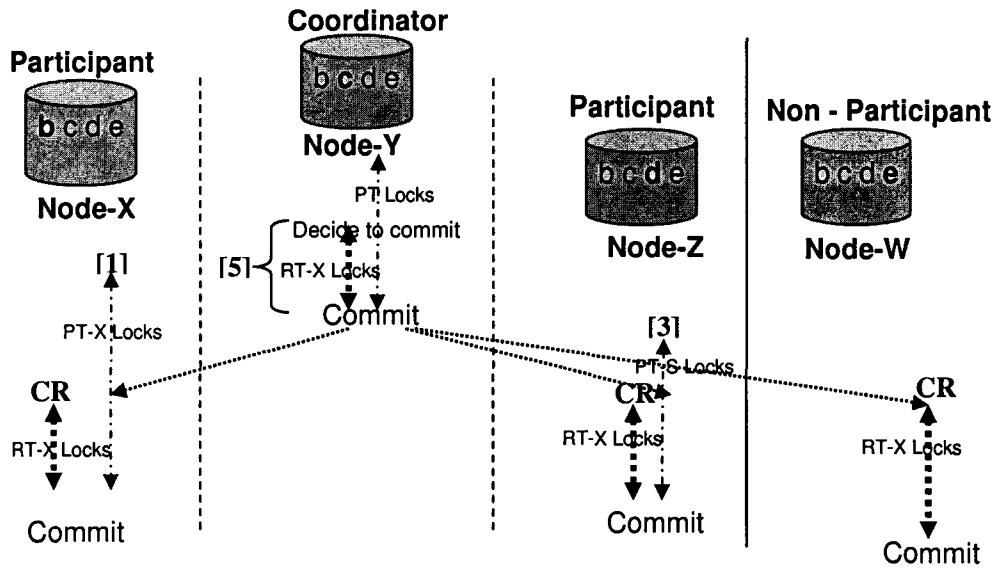

Figure 4.4 Illustrates the execution of the refresh transaction protocol along with the primary transaction protocol

For Coordinator, Participants and Non-Participants:

The execution of a refresh transaction is the same for all the roles. Let us assume that the local TM uses 2PL for execution of refresh transactions. Consider the execution of the refresh transaction of T_K at Node-X.

1. Lock acquisition phase: RT-X locks are acquired on all the data items to be written.
2. Execution phase: The transaction is executed. Now, values of T_K's data items in its write-set present at that node are written.

3. Commit phase: Transaction commits. RT-X locks on all the data items of T_K are released.

In Example 4.1 (refer Figure 4.4), after the primary transaction decides to commit, the refresh transaction protocol is executed first at the coordinator node. This is indicated by [5], as it is performed at stage (5) in coordinator's view of the primary transaction. Then, the change-records are sent to all the nodes. The change-records are implemented at these nodes using the refresh transaction protocol. It should be noted that at the participant nodes, the primary transaction locks and the refresh transaction locks (ie PT-X and RT-X locks) co-exist, whereas at the non-participant nodes, only the refresh transaction locks are acquired. The symbol [1] at Node-X and [3] at Node-Z indicates that these locks were obtained in stage (1) and stage (3), respectively, of the primary transaction protocol in its participant's view. In Figure 4.4, CR indicates that these RT locks were acquired after the delivery of the change-record message.

4.1.3 For ROT T_K

Usually, the locking mechanism is used for execution of the ROT protocol. Its execution has three stages:

(1) Lock acquisition phase

(2) Execution phase

(3) Commit phase

A ROT can be executed at any node. Let us consider the execution of a ROT at Node-

X.

(1) Lock acquisition phase: ROT-S locks are requested on all the data items at the present node. The ROT protocol is compatible with the execution of the primary transaction protocol but not compatible with the execution of the refresh transaction protocol. (That is, ROT-S locks are compatible with PT locks but not with RT locks.)

(2) Execution phase: The transaction is executed.

(3) Commit phase: the transaction committed and all the ROT-S locks are released.

In Figure 4.5, Node-X executes ROT T_R when there is a PT lock on data item b. This shows that ROTs and primary transactions can be executed concurrently. But, ROTs and refresh transactions of conflicting transactions cannot be executed concurrently.

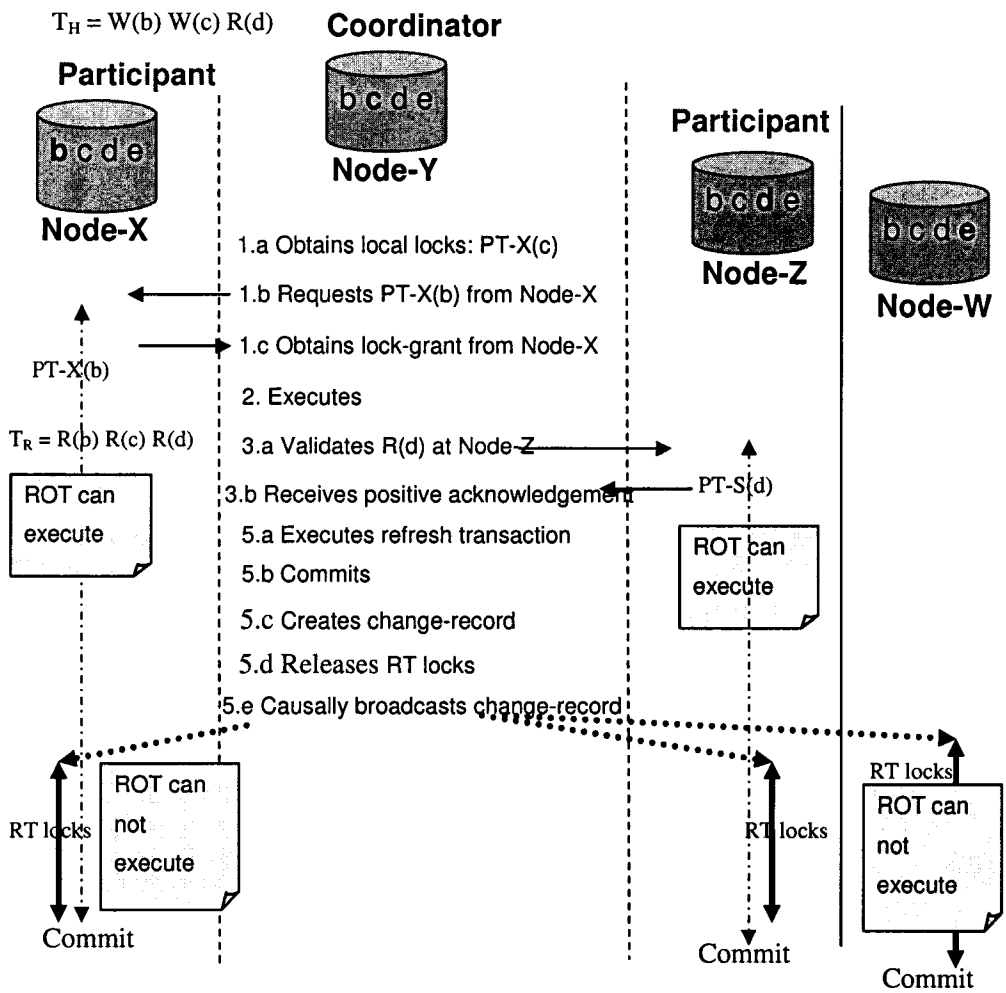

Figure 4.5 The execution of the primary transaction protocol to illustrate the execution of ROTs

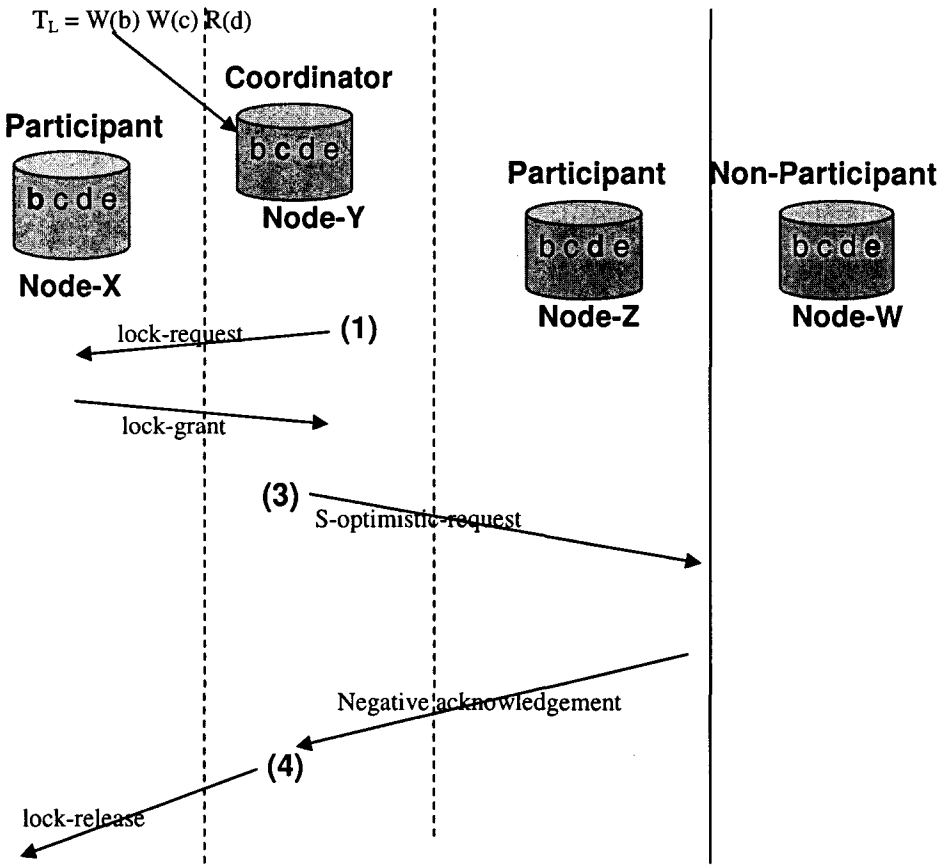

Figure 4.6 Illustrates the messages exchanged between the nodes (in terms of roles) for the execution of T_L

Figure 4.6 illustrates all the messages exchanged between the nodes. Consider the transaction, T_L, executing with Node-Y as its coordinator. The transaction T_L requests a PT-X lock on the data item b in stage (1). Then, it sends an S-optimistic-request message to validate the operation on data item d in stage (3). T_K is aborted due to the negative acknowledgement. So, it sends lock-release messages to the other participants.

4.1.4 Session guarantee for transaction T_K

Session P, in which a user interacts with the registry, is represented by the two dimensional array S_P[1,2,3,...,n][1,2,3,...,n] (denoted by S-array). Informally, S-array contains the state of the registry previously seen by the user. A transaction, submitted to the system by the input stream, contains the S-array indicating the user's session. When a new ROT in session P is scheduled for execution, the *session_read* procedure is first executed. This procedure reads data from the S-array of session P (That is, S_P[1,2,3,...,n][1,2,3,...,n]). After the ROT is executed, S-array is updated by the *session_update* procedure. The procedure stores the state of the system as seen by the operations of the session. This procedure is also performed after the execution of the primary transaction of the update transaction.

S_P[1,2,3,...,n][1,2,3,...,n] contains the set of USNs of the transactions seen by the user session P (directly or indirectly) in its previous operations.

Example 4.3: Consider the setup as shown in Figure 4.7. The state of Node-X can be represented using Nx[x][1,2,3,...,n] indicated by the arc AB in the figure. Arc AB indicates all committed transactions at Node-X (similarly arcs CD, EF, and GH represent for the other nodes). The maximum state of the registry is represented by the set of the latest committed transactions in the system. The minimum state of the registry is represented by the set of transactions which have committed at their coordinator nodes and yet to be executed at some other nodes in the registry. As the registry is a loosely synchronized system, no node may know the exact minimum and

maximum states of the registry. Therefore, the state of the node is stale compared to the maximum state of a system in its one copy equivalence. Minimum state of the registry in its one copy equivalence is inferred from minimum states of all nodes. The actual minimum and maximum states of the registry are not stored any where. A user may infer maximum and minimum states of the registry when he executes ROTs at different nodes in the registry.

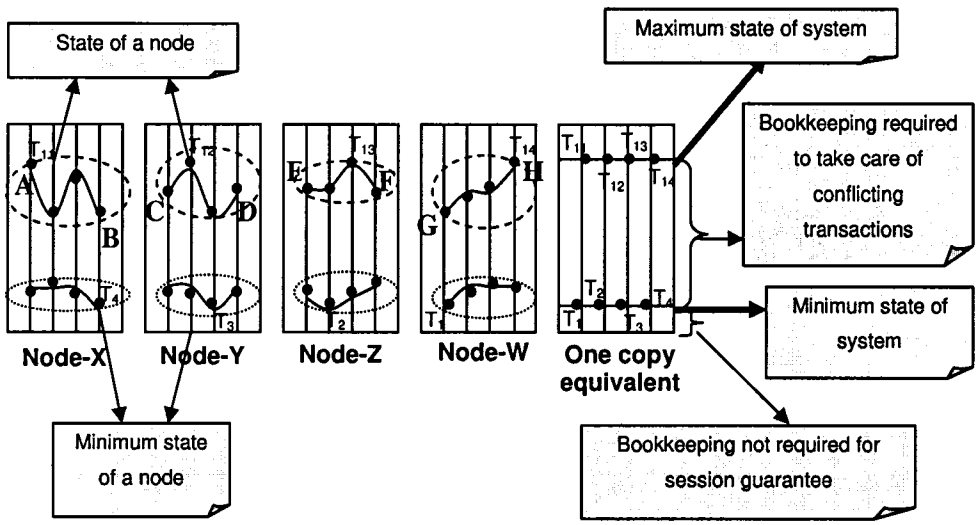

Figure 4.7 Illustrates the minimum and maximum states of each individual nodes along with its one copy equivalent

The minimum and maximum states of the registry are represented by the set of transactions T_1, T_2, T_3, T_4 and T_{11}, T_{12}, T_{13}, T_{14}, respectively. Node-X has executed transaction T_{11} which is the latest in the system. Similarly, the latest transactions in the system executed at Node-Y, Node-Z, and Node-W are T_{12}, T_{13}, and T_{14},

respectively. T_1 at Node-W represents the oldest transaction which is yet to be executed at some node in the registry. It may also be the case that T_1 has been executed at all the nodes but Node-W is not aware about it. T_1 forms a part of the minimum state of the registry. Similarly, T_2, T_3, and T_4 also constitute the minimum state of the registry. For Node-X, $N_X[x][1,2,3,\ldots,n]$ represents its state while $M_X[1,2,3,\ldots,n]$ represents its minimum state. The array representing the state of the node and maximum state of the node are represented by N-array and M-array, respectively.

We consider two procedures for maintaining the session guarantee mechanism – *session_read* and *session_update*. The former procedure is used before execution of a ROT, while the latter is used after the execution of all types of transactions. The *initialization* and *delete_change-record* procedures are used by the registry to book keep the storage requirement and for message propagation mechanism.

Initialization:

When the system starts, M-array stored at each of the node in the registry is initialized to zero value. M-arrays at Node-X, Node-Y, Node-Z, and Node-W are initialized as follows:

$M_X[1,2,3,\ldots,n][1,2,3,\ldots,n] := M_Y[1,2,3,\ldots,n][1,2,3,\ldots,n] := M_Z[1,2,3,\ldots,n][1,2,3,\ldots,n] := M_W[1,2,3,\ldots,n][1,2,3,\ldots,n] := 0$

This assignment means that each element of the two dimensional array is assigned zero value. When a new session P starts in the system, its S-array is initialized to zero value.

$S_P[1,2,3,...,n][1,2,3,...,n] := 0$

Session guarantee procedures:

A simple solution to ensure session guarantees is to update S-array of session P with the state of a node at which the transaction was executed, after its execution. Then, the next transaction in session P is executed only after that recorded state in S-array is reached at the executing node. As there is no common sequence of state changes at all the nodes, wait time may increase unpredictably between consecutive transactions. Our fine grained session guarantee mechanism aims at reducing wait time and making it more predictable.

Session_read:

The update transaction always accesses the latest values in the registry, as otherwise 1SR cannot be ensured. But, ROTs can read any values between minimum and maximum states of the system (in one copy equivalence). In order to provide the consistent database view to the user, our protocol ensures Read Only Follow Updates and Monotonic Read Only guarantees. All the ROTs should perform the *session_read* procedure before executing the transaction.

The latest set of transactions of session P which have updated or seen the updates of data items in custody of Node-X is given by $S_P[x][1,2,3,\ldots,n]$. Similarly, the latest set

of data items in custody of Node-X and Node-Y seen by the user session is obtained by computing the recent transactions among $S_P[x][1,2,3,\ldots,n]$ and $S_P[y][1,2,3,\ldots,n]$. In general, for a ROT reading data items in custody of Node-X, Node-Y, and Node-Z, the following strategy is used. (We assume some basic name service directory to find out custodians of data items.) In the *session_read* procedure, first, finds transactions already seen by session P in custody of these nodes. Later, the ROT is executed only if these transactions have been already executed at Node-Q. In summary, the procedure computes the USNs of the conflicting transactions which at least must have been executed by reading values from session variable (in user input stream). Then, it waits only for those transactions.

```
 // Let Temp be an one dimensional array of size n
for var = 1 to n, where n is the number of nodes
 temp[var] := Max(SP[x][var], SP[y][var], SP[z][var])
 where Nodes x, y, z are the custodians of data items of the ROT
wait until ( NQ[q][1,2,3,...,n] ≥ Temp[1,2,3,...,n] )
  execute the ROT at Node-Q
```

First three lines of the above procedure computes and stores in Temp array the latest transactions reading or updating data items in custody of Node-X, Node-Y, and Node-Z in session P. The transaction can read the data item only if the state of the node is greater than state of the system stored in the Temp array. Otherwise, it waits until that state is reached. As a new ROT to be executed waits for only the most relevant transactions, the unnecessary wait time is minimized.

Session_update:

We have two different kinds of procedures for *session_update*. One kind is used for update transactions, and the other is used for read only transactions. The former kind of procedure ensures the Read Only Follow Updates session guarantees while the latter one ensures Monotonic Read Only session guarantees.

The *session_update* procedure used for an update transaction can be explained as follows:

Let the update transaction, T_K, be executed in session P. Then, the following procedure updates the session variable (in user input stream) with the USNs of transactions that conflict with T_K and of T_K.

```
for var = 1 to n, where n is the number of nodes
  for varp = 1 to n, where n is the number of nodes
    SP[var][varp] := Max(DK[var][varp], SP[var][varp])
```

In the above code, please notice that there is one to one correspondence between D-array of the update transaction and S-array of the user session. D_K contains USNs of all the oldest transactions directly or indirectly conflicting with T_K, including itself. USNs of all the conflicting transactions in their D-arrays are copied to S_P. Now, S_P contains the oldest transactions recently seen by the user so far. Note that S_P of a session contains transactions seen and not the present state of the node. Therefore,

session guarantee mechanism is fine grained and increases the performance of the system.

The *session_update* procedure used for a ROT can be explained as follows:

A ROT may have read data items written by more than one transaction. The ROT conflicts with all these transactions. The TM lists these conflicting transactions. (The conflicts are found using the same procedure, as used in the creation of the D-array.) Let D-arrays corresponding to these conflicting transactions be D_1, D_2, D_3,..., D_M. Then the following procedure updates the session variable with USNs of transactions conflicting with T_K.

for var = 1 to n, where n is the number of nodes
 for varp = 1 to n, where n is the number of nodes
  S_P[var][varp] := Max(D_1[var][varp], D_2[var][varp], D_3[var][varp],...., D_M[var][varp], S_P[var][varp])

The above code computes and stores in S_P, all the conflicting update transactions seen by the ROT. In both the above kind of procedures, after a transaction commits the result of the execution is returned to the user. The next transaction in session P cannot proceed until the *session_update* procedure is executed and the user receives the response.

Delete_change-record:

As change-records generated in the registry are stored at all the nodes, they grow without bound. Using this procedure, we delete the change-records at nodes so that

the space required for bookkeeping can be reduced. If the primary transaction of T_K is executed at Node-X and is successfully committed, we know that at some point in time, all the nodes in the registry will execute T_K. $M_X[1,2,3\ldots,n]$ array indicates to Node-X, a set of USNs of transactions that have been executed at Nodes 1,2,3,…,n in the registry. The procedure is designed such that the change-records of these transactions can be deleted at Node-X and still session guarantee can be ensured. The *Delete_change-record* procedure computes and stores in M-array, the USNs of transactions that have been executed at all the nodes as follows:

```
for var = 1 to n, where n is the number of nodes
  MX[var] := NX[var][var]
 //stores Node-X's knowledge about the last primary transaction executed at Node-var
  for varp = 1 to n, where n is the number of nodes
   MX[var] := Min( NX[varp][var], MX[var])
//calculates the transaction which has been executed at all nodes whose primary
//transaction was executed at Node-var
   where the Min function returns the minimum integer of a set of integers.
```

The above code, computes and stores in $M_X[y]$ the latest primary transaction that was executed at Node-Y and has been executed at all the nodes. Therefore, M_X, computed at Node-X, contains the set of USNs of primary transactions which were executed at different nodes and has been executed at all the nodes known by Node-X. *Delete-change-record* procedure is invoked by Node-X whenever any of the entries in $M_X[1,2,3,...,n]$ array is updated. As change-records are ordered with the latest on top and oldest at the bottom, the procedure searches for the change-record with identifier $<i, M_X[i]>$ from bottom to top and deletes that change-record.

The space required to store change-records at a node is directly dependent upon its knowledge about transactions that have already been executed at other nodes indicated by the M-array at that node. Therefore, sending the P-array with all the message types increases the freshness of the M-array. If the cost of communication between the nodes is higher than the cost of space, then the P-array is sent with only some of the messages. On the other hand, if the cost of space required is higher than the cost of communication between the nodes, then the P-array is sent along with all the messages. Basically, the system has flexibility to decide how frequently to send the P-array.

4.2 Causal transmission of messages

Our protocol facilitates causal transmission of messages. If a message has to be received in causal order, it should contain the D-array. The only messages having D-arrays are – change-record, lock-grant, and acknowledge messages and these are delivered in causal order. (Lock-request, X-optimistic, S-optimistic, and lock-release-request messages are not delivered causally.) The procedure to ensure causal message delivery is as follows:

Consider the message of transaction T_K sent from Node-Y to Node-X. Then, the procedure waits until all conflicting transactions of T_K have been delivered at Node-X and later updates the state array at Node-Y to indicate the delivery event as follows:

```
Initialize D_K-COL[1,2,3,...,n] := 0
// All the elements of array are initialized to zero value
for var = 1 to n; where n is the number of nodes
 for varp = 1 to n; where n is the number of nodes
  D_K-COL[var] := Max(D_K-COL[var], D_K[varp][var])
  //converts two dimensional array to one dimensional array
Wait until N_X[x][1,2,3,...,y-1,y+1,...,n] ≥ D_K-COL[ 1,2,3,...,y-1,y+1,...,n]
 then deliver T_K at Node-X.

//After delivery of the message, N_X[1,2,3,...,n][1,2,3,...,n] is updated
if the change-record contains P_Y
 N_X[x][y] := Max(N_X[x][y], P_Y[y])
  for var = 1 to n; where n is the number of nodes
   N_X[y][var] := Max(N_X[y][var], P_Y[var] )
else
  N_X[x][y] := Max(N_X[x][y], D_K[y][y])
   for var = 1 to n; where n is the number of nodes
    N_X[y][var] := Max(N_X[y][var], D_K[y][var] )
```

In the above code, the two dimensional array D_K is converted to the one dimensional array $D_{K\text{-}COL}$. This array is compared with the state array $N_X[x][1,2,3,\ldots,n]$. The transactional message is delivered only if its state is greater than or equal to the state of Node-X. It should be noted that in our algorithm, the causal delivery of message of T_K at Node-X involves comparison of the state array of Node-X and the D-array of T_K. That is, a message delivery will be delayed only if its preceding conflicting messages are not delivered. This imposes a total order delivery between two messages only if they are conflicting. This also eliminates the false causality (the perception that because one event occurred before another, the earlier event has caused the later event). Also, note that message need not contain the P-array to be delivered causally.

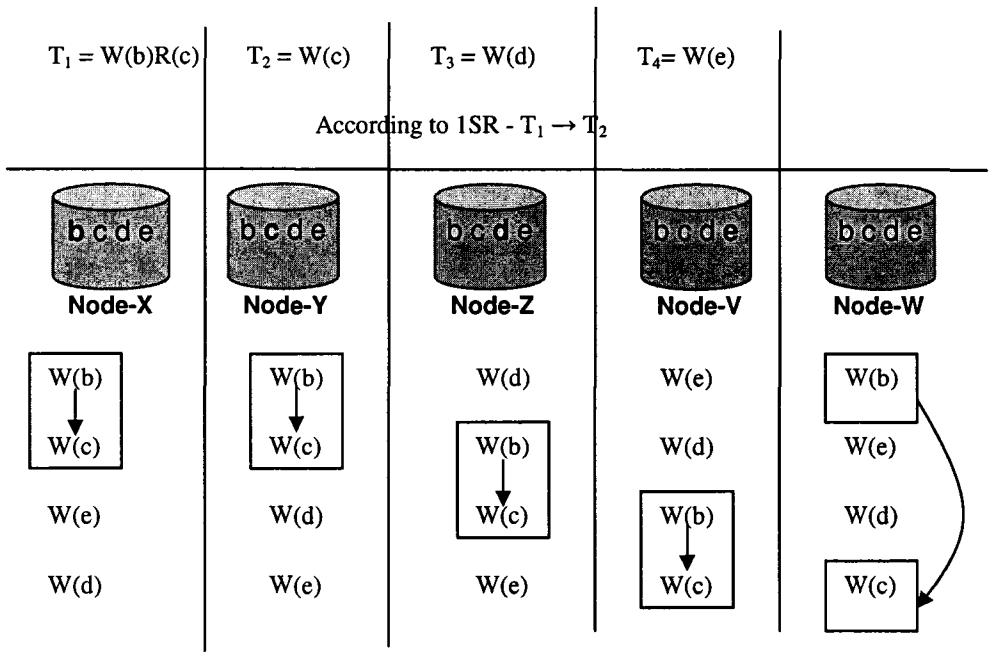

Figure 4.8 Illustrates the total order of delivery and execution of W(b) and W(c) corresponding to T_1 and T_2, respectively

Example 4.4: In Figure 4.8, T_1, T_2, T_3, and T_4 are executed at Node-X, Node-Y, Node-Z, and Node-V, respectively. It should be noted that T_1 and T_2 are conflicting transactions (WR conflict). Therefore, the causal delivery mechanism ensures a total order among messages of these transactions. It provides the flexibility to execute T_3 and T_4 in any order at all the nodes.

4.3 Fully optimistic replication protocol

The replication protocol facilitates the coordinator to execute a transaction either pessimistically or optimistically. That is, a lock on a data item for a global transaction

may be requested either before or after its execution. We know that any non-conservative locking mechanism is prone to the occurrence of deadlocks. Distributed deadlocks are possible in our protocol as locks cannot be requested atomically. (That is, locks required by the transaction may be local locks, pessimistic global locks and optimistic global locks. Each of these kinds of locks is requested atomically. But all these different kinds of locks together cannot be requested atomically.)

In the replication protocol discussed in section 4.1, the optimistic lock request on a data item is performed after its execution; but validated before the broadcast of the change-record. The protocol can be modified to be fully optimistic by deferring the lock request on the data item until the change-record message is delivered at a node similar to [HSAE03]. Thus, deadlocks can be prevented in our system. The fully optimistic replication protocol can be explained as follows:

The primary transaction is executed at the coordinator node. If the transaction is executed without requesting a lock on at least one data item until the change-record is broadcasted, then it is said to be in a pre-committed state. The change-record of the pre-committed transaction is broadcast to all the nodes. If a conflict is found at any of the participant nodes, then the transaction is aborted. Otherwise, on the successful commitment at all participant nodes, the transaction commits at all the nodes in the system. Please note that for the local transactions the protocol is same as explained earlier.

The fully replicated protocol can be explained from the coordinator, participants, and non-participant's views.

For Coordinator (Coordinator's view)

Consider the execution of the primary transaction, T_K, at the coordinator, Node-Y.

(1) Lock acquisition phase: Requests PT-S and PT-X locks on the data items to which Node-Y is custodian, atomically. That is, it obtains locks on all the data items in custody of Node-Y or does not obtain any lock. (Please note that the locks at the remote node are not requested in this phase.)

(2) Execution phase: Transaction T_K is executed. T_K pre-commits. This state of T_K is known as pre-committed state. All writes are performed in the private workspace.

(3) Broadcast of the change-records of the transaction: The change-records of the transaction is created and broadcast to other nodes. The creation of the change-record is as follows:

   a) The USN is generated.

   b) Then, updates state array N_Y and creates the present state array P_Y, the D-array D_K.

   c) Creates change-record for T_K.

   The procedures to create these arrays are similar to one explained in section 4.1 except that for creation of the D-array for a pre-committed transaction, the preceding conflicting pre-committed transactions are not considered (that is, only committed transactions are considered). The change-record of a pre-committed

transaction is indicated as the pre-committed transaction. Please note that in the original replication protocol, the change-record is used only for a committed transaction.

(4) Commit phase: Whenever the N-array at Node-Y is modified, the following check is performed. The Node-Y checks if the change-record of T_K has been successfully delivered and pre-committed at all the participant nodes of T_K. That is, if Node-X, Node-Z, and Node-V are the other participants of T_K and tusn is the USN of T_K. Then, the following procedure checks, if T_K has been delivered at participant nodes. On successful delivery of change-records at all the nodes, the refresh transaction protocol is executed and transaction commits.

Wait until ($N_Y[x][y] \geq$ tusn && $N_Y[z][y] \geq$ tusn && $N_Y[v][y] \geq$ tusn)
then executes the refresh transaction protocol of T_K and commits

In the above code, T_K at Node-Y waits until its state array is updated with the USN of T_K, indicating its delivery at Node-X, Node-Z, and Node-V. Then, T_K commits.

After T_K commits, the following steps are performed:

a) Releases all the PT locks.

b) Converts the pre-committed change-record of T_K stored at that node to a (regular committed) change-record.

(5) Abort phase: When a pre-committed transaction is delivered at a node, it checks if T_K conflicts with any other pre-committed transactions. If any conflict is found, then both the transactions are aborted. All PT locks of T_K are released. Then, change-record message of T_K is deleted.

For Participants (participant's view):

Consider the execution of the primary transaction, T_K, at the participant, Node-X.

(1) Upon receiving the pre-committed change-record message (refer to stage (3) in the coordinator's view): On receiving the change-record of the pre-committed transaction, T_K, the protocol checks for the conflicting pre-committed transactions. If there are no conflicting pre-committed transactions, then PT locks are obtained on the data items of T_K, to which Node-X is custodian of. If the locks are held by the conflicting transactions, then the protocol goes to stage (3), where T_K is aborted. It should be noted that checking for the conflicts and obtaining locks are performed as an atomic operation.

(2) Commit phase: Whenever the N-array at Node-X is modified, Node-X checks, if the change-record of T_K has been successfully delivered and pre-committed at all the other participant nodes of T_K. If the outcome is true, the refresh transaction protocol for T_K is executed and T_K commits. Then, the following steps are performed:

a) Converts the pre-committed change-record of T_K stored at that node to a committed change-record.

b) Releases all the PT locks.

(3) Abort phase: When a pre-committed transaction is delivered at a node, the protocol checks if T_K conflicts with any other pre-committed transactions. If any conflict is found, then both the conflicting transactions are aborted. All PT locks of T_K are released. Then, change-record message of T_K is deleted.

For Non-Participants (Non-Participant's view):

Consider the execution of the primary transaction, T_K, at the participant, Node-W.

(1) Upon receiving the change-record message of pre-committed transaction (refer to stage (3) in the coordinator's view): On delivery of the change-record of the pre-committed transaction, T_K, the protocol checks for the conflicting pre-committed transaction. If any conflict is detected, it goes to stage (3) where it is aborted.

(2) Commit phase: T_K commits at Node-W, if the change-record of T_K has been successfully delivered and pre-committed at coordinator and all the participant nodes. Then, the following steps are performed:

a) Executes the refresh transaction protocol for T_K.

b) Converts the pre-committed change-record of T_K stored at that node to a committed change-record.

(3) Abort phase: If Node-W receives the abort message from another node, then the pre-committed transaction is aborted and deletes the change-record message.

Read only transactions

During the creation of the D-array for the ROT (for session guarantees), the pre-committed transactions are not considered.

The main disadvantage of the fully optimistic protocol is that, as the conflict rate increases, the abort rate also increases and the system performance decreases.

4.4 Correctness Proof

First, we show that all the dependent messages are delivered causally without the effect of false causality. Then, we show that the primary transaction protocol ensures 1SR. Also, we show that the session guarantee mechanism ensures Read Only Follow Updates and Monotonic Read Only guarantees. Finally, we show that the fully optimistic protocol ensures one copy serializability.

Lemma 1: If T_K conflicts with T_L where T_K precedes T_L ($T_K \rightarrow T_L$), then either the D-array of T_L contains the identifier <Node-ID, USN> of T_K or the latest transaction.

Proof: Let T_K and T_L be executed with Node-X and Node-Y, respectively, as their coordinators. Let us assume that the change-record of T_K is present at Node-Y. Also, let T_K be the immediately preceding transaction of T_L. As for each operation in T_L, the procedure checks if there is a conflicting operation in any of the change-records, starting from the present state to the oldest state of the node, it will discover the change-record of T_K. Any of the WW, WR, and RW conflicts between the transactions will be detected.

Consider the following code which creates D-array for T_L.

```
for var = 1 to n; where n is the number of nodes
 for varp = 1 to n; where n is the number of nodes
  DL[var][varp] := Max(DK[var][varp],DL[var][varp])
```

In the above code, as T_K precedes T_L, creation of D-array takes maximum USNs of D_K and D_L to find the latest conflicting transactions to T_L. This assignment ensures that $D_L[x][x]$ holds the USN of T_K.

Let us assume that T_K is not the immediately preceding transaction. Let T_G be immediately preceding transaction to T_L and T_K precedes T_G. Therefore, $T_K \rightarrow T_G \rightarrow T_L$. By the above procedure, when D_G is calculated, it contains the USN of T_K. Similarly, when D_L is calculated the procedure takes the maximum USNs from each of the nodes. D_L will have the identifier of T_K or the succeeding transaction T_G from that node.

On other hand, if the change-record of T_K has been deleted by *delete_change-record* at the coordinator node of T_L, it means that T_K has been executed at all nodes already. As T_K is already delivered at all nodes, there is no need for T_L to wait for message of T_K. In such a situation, even if T_L does not have the USN of T_K, it does ensure a correct execution.

The proof is complete.

Theorem 1: The broadcast mechanism ensures causal delivery of messages of conflicting transactions.

Proof: Let Φ_K and Φ_L be the messages corresponding to T_K and T_L where T_K precedes T_L ($T_K \rightarrow T_L$). This precedence relation is induced when D-array for Φ_K and Φ_L are created. Let Φ_K and Φ_L be created and sent by Node-X and Node-Y, respectively.

Consider the delivery of Φ_K and Φ_L at Node-Z.

Now, let $N_Z[z][1,2,3,...,n]$ be the state array of Node-Z. A message of the transaction can be either of a change-record or a lock-grant/acknowledge message. Thus, for the conflict between T_K and T_L, we have the following types of messages in the system:

- If both Φ_K and Φ_L are change-records, then $T_K \rightarrow T_L$ means that $D_L[1,2,3,\ldots,n]$ contains at least the USN of T_K (Φ_K) or the latest transaction (by Lemma-1).....(i)
- If both Φ_K and Φ_L are lock-grant/acknowledge messages, $D_L[1,2,3,\ldots,n]$ contains the USN of T_K (Φ_K) which is the same as case (i).
- If Φ_L is a lock-grant/acknowledgement message, then Φ_K may be the change-record of its preceding conflicting transaction. Then, this case reduces to the same as in (i).
- If Φ_K is a lock-grant/acknowledge message on some data item, then its succeeding conflicting transaction T_L cannot obtain the lock on that data item until T_K finishes its execution. So this case is invalid.

Effectively we have only one case namely, case (i) where the succeeding transaction's message type contains the preceding transaction's identifier.

We prove the theorem with the following two claims:

(1) All the conflicting transactional messages are delivered causally.

(2) A message will be eventually delivered after a bounded period of time.

Claim 1: All the conflicting transactional messages are delivered causally.

Consider the delivery of Φ_L at Node-Z which is sent from Node-Y. The delivery of Φ_L means that the following condition is satisfied.

'$N_Z[z][1,2,3,\ldots,y-1,y+1,\ldots,n] \geq D_{L\text{-}COL}[1,2,3,\ldots,y-1,y+1,\ldots,n]$'

$D_{L\text{-}COL}[1,2,3,\ldots, y-1,y+1,\ldots,n]$ contains the latest USNs of the transactions conflicting with T_L whose primary transaction was executed with Nodes $1,2,3,\ldots, y-1,y+1,\ldots,n$ as the coordinator, respectively. Satisfying the above condition means that the state vector of Node-Z ($N_Z[z][1,2,3,\ldots,n]$) contains the USN of T_K (by Lemma 1). That is, Φ_K would have been delivered already. Until Φ_K is delivered, Φ_L cannot be delivered. Hence, causality is maintained.

Claim 2: A message will be eventually delivered after a bounded period of time.

On the contrary, let us assume that there is a set of messages of transactions sent from Node-Y to Node-Z which is not delivered. Let Φ_L be first among such messages. Let T_K precede T_L in the serialization graph. These transactions can be either a local transaction or a global transaction. If both are local transactions whose primary transaction executed at Node-Y, then the reliable broadcast mechanism ensures that they are delivered in the same order at all the nodes. On the other hand, if one of them is a global transaction, then the eventual delivery must be ensured.

Let us assume that T_K and T_L are global and local transactions, respectively. Let the primary transactions of T_K and T_L be executed at Node-X and Node-Y, respectively. By Lemma 1, $D_L[x][1,2,3,\ldots,n]$ of Φ_L contains the USN of T_K. If Φ_L is not delivered at Node-Z, the following condition is not satisfied.

"$N_Z[z][1,2,3,\ldots,y\text{-}1,y+1,\ldots,n] \geq D_{L\text{-}COL}[1,2,3,\ldots y\text{-}1,y+1,\ldots,n]$"

This means that the state of Node-Z waiting for the delivery of Φ_K. But due to reliable broadcast of message, sent from Node-X Φ_K must be delivered at Node-Z after finite amount of time. This is contradictory to our assumption that Φ_L is not delivered waiting for other messages. Using the same reasoning, it can be shown that even if both are global transactions, then T_K will be delivered.

The proof is complete.

Lemma 2: Lock management obeys two phase locking (2PL) at each node.

Proof: For the 2PL to hold true at a node, once a transaction starts releasing locks at a node, it should not acquire any further locks. In our protocol, locks are released in stage (4) or (5) of the primary transaction protocol in coordinator's view and stage (3) of the refresh transaction protocol. We show that every transaction executed at any node follows the 2PL. (A few of the transactions may have RT-X locks and PT-X locks on the data items simultaneously. These locks are held till the commit point and no lock is requested after this point.)

Case 1: For the refresh transactions (executed using refresh transaction protocol).

Once the transaction reaches stage (3), it does not request any further locks. Only after execution, the transaction starts to release the locks. Hence, the refresh transaction follows 2PL.

Case 2: For the primary transaction (executed using the primary transaction protocol). All locks in the coordinator's view are acquired at stage (1) or stage (3). They are released at stage (4) or (5). If they are released in stage (4), the transaction is aborted. If the transaction has to commit successfully, the primary transaction has to release the locks in one of the following stages:

- Stage (5) of the coordinator's view.
- Stage (3) of the participant's view.

The stage (3) of participant's view occurs only after the lock-grant/acknowledgement message is given for the primary transaction in stage (2) in participant's view. Locks at participant nodes are not requested after stage (2) in participant's view. Also, the refresh transaction is nested inside the primary transaction protocol before releasing its locks.

As from case 1, the refresh transaction protocol follows 2PL and no locks are requested after the refresh transaction protocol starts executing, in both the above cases.

For the coordinator's view, it is clear that when lock releasing starts, the primary transaction should have already acquired all the required locks as stage (5) succeeds stage (3) at the coordinator.

For the participant's view, the primary transaction releases the lock at its stage (3). The coordinator would have acquired lock at this participant's node in stage (1) or stage (2).

Therefore, the primary and refresh transactions ensure 2PL at both the coordinator and participant nodes. In summary, if a transaction starts releasing locks, it must have aborted or it does not request any further locks. Therefore, all the update transactions follow 2PL.

Case 3: For ROT

It acquires all the required locks in the beginning. Then, executes the transaction. It does not request any locks after stage (3). Hence ROTs follow 2PL.

The proof is complete.

Lemma 3: Local lock management ensures conflict serializability

Proof: In Lemma-2 we have shown that for a particular transaction, 2PL is ensured at the local nodes. It follows from [BGH87] that all the transactions are conflict serializable. The proof is complete.

Theorem 2: At a global level acyclic Global Serialization Graph (GSG) is obtained for all the update transactions.

Proof: Let T_K and T_L be conflicting transactions. Let T_K precede T_L ($T_K \rightarrow T_L$). We have to show that the two transactions are totally ordered.

First, we consider a case where both these transactions have the same coordinator node. The transactions may conflict on data items in custody of the common coordinator node or at another node. Later, we consider both transactions having different coordinator nodes. Here also, transactions may conflict on a data item in custody of one of these coordinator nodes or at other nodes.

Case 1: Transactions have the same coordinator node and conflict on a data item, which is in custody of the common coordinator node.

Both T_K and T_L request locks from the same local TM. By Lemma 3, the local TM serializes these two transactions. Change-records are created in the same order and sent to other nodes in the same order (Theorem-1). Eventually, they are executed in the same order at all those nodes. These two transactions are globally serializable.

Case 2: Transactions have the same coordinator node and they conflict on a data item, which is in custody of a node other than the common coordinator node.

Both T_K and T_L request the locks from the remote TM. The remote TM gives the lock to T_K first and then to T_L (because, $T_K \rightarrow T_L$). T_L obtains the lock from the remote TM only after the change-record of T_K is received and executed at the remote node as a refresh transaction. When the remote TM sends the lock-grant message to T_L, the message's D-array contains the USN of T_K (identified by, <Node-ID of coordinator of T_K, USN of T_K>). This dependency is copied later in $D_L[1,2,3,...,n][1,2,3,...,n]$ of T_L's change-record. By Theorem-1, these change-records are delivered in this order at

all the nodes. Also, they are executed in this order at all the nodes. These two transactions are globally serializable.

Case 3: Transactions have different coordinator nodes and they conflict on a data item, which is in custody of one of the coordinator nodes.

Let the conflict be at T_K's coordinator. T_L will obtain the lock from the TM at that node only after T_K's change-record is created and locks are released. Lock-grant message that T_L obtains will have the D-array containing the USN of T_K. Therefore, the D-array of T_L's change-record will have the USN of T_K. By Theorem-1, they are totally ordered at all the nodes. These two transactions are globally serializable.

Case 4: Transactions have different coordinator nodes and they conflict on a data item, which is in custody of neither of these coordinator nodes.

Let both of these transactions conflict on a data item at a remote node. The TM at the remote node first grants the lock to T_K. Later, T_L obtains the lock only after the remote TM receives the change-record of T_K and the refresh transaction is executed at that node. The lock-grant message's D-array for T_L will contain the identifier of T_K. The change-record of T_L will contain the identifier of T_K in its D-array (by lemma-1). By Theorem-1, these two change-records are totally ordered. Hence, they are globally serializable.

In all the above cases it is assumed that the coordinator will use pessimistic method to request locks. Even if any of the coordinators request the lock optimistically, the

proof remains the same, as both types of messages contain information about the all preceding conflicting transactions in their D-arrays.

The proof is complete.

Theorem 3: Session guarantee is ensured.

Proof: Let us assume that two update transactions are executed in the system in such a way that T_K precedes T_L ($T_K \rightarrow T_L$). Let T_K and T_L update data items b and c, respectively. Please note that T_K and T_L may also update data items other than b and c, respectively. But our interest here is only on the updates of data items b and c. Let Node-X and Node-Y be the coordinator and the participant, respectively, of T_K. Let Node-W and Node-Z be the coordinator and the participant, respectively, of T_L. Let user session P execute transactions T_G followed by T_H. T_G reads data item c at Node-U written by T_L. Now, when T_H reads data item b at some node, we have to show that it will read, at least, the version written by T_K.

Let the identifier of T_K and T_L be <i, x> and <j, w>, respectively.

During creation of the change-record of T_K, D_K is obtained as follows:

$D_K[x][x] := D_K[y][x] := i$

During creation of change-record of T_L, D_L is obtained as follows:

$D_L[x][x] := D_K[x][x]$ (value is i, because of conflict with T_K)
$D_L[y][x] := D_K[y][x]$ (value is i, because of conflict with T_K)
$D_L[w][w] := D_L[z][w] := j$ (after updating present transaction)

When session $S_P[1,2,3,...,n][1,2,3,...,n]$ executes its first transaction T_G, it reads from T_L. The D-array of T_L is copied to S_P. When the next transaction, T_H, is submitted to Node-Q, it is executed only if the *session_read* procedure satisfies the condition - $N_Q[q][1,2,3,\ldots,n] \geq S_P[x][1,2,3,\ldots,n]$ (actually, $N_Q[q][1,2,3,\ldots,n] \geq$Temp $[1,2,3,\ldots,n]$)

In the code $S_P[x][1,2,3,\ldots,n]$ is used, as the data item to be read is in custody of Node-X. Satisfying this condition means N_Y has executed transaction T_K. Therefore it will read the transaction written by at least T_K.

The proof is complete.

Theorem 4: The fully optimistic protocol ensures one copy serializability.

Proof: Let T_K and T_L be conflicting transactions. Let T_K precede T_L ($T_K \rightarrow T_L$). We have to show that the two transactions are totally ordered.

If both T_K and T_L are local transactions, then the local TM will ensure that these transactions are executed in the order; T_K precedes T_L, and later delivered and executed in the same order at all the other nodes.

Let us consider global transactions. Let us assume that the primary transactions of T_K and T_L be executed at Node-X and Node-Y, respectively. Also, let T_L be a global transaction, accessing at a data item in custody of Node-X, due to which it conflicts with T_K. After T_L is executed, it is said to be in a pre-committed state. The change-record of T_L is sent to Node-X. At Node-X, if no conflicts are found on the data item either due to a pre-committed transaction or a committed transaction, then the

change-record of T_L is delivered at that node. Only after Node-Y knows that T_L is delivered successfully and no abort message is received, T_L commits. Therefore, T_K and T_L are totally ordered. Similarly, we can show that T_K and T_L are totally ordered at all nodes for other combination of global transactions.

The proof is complete.

4.5 Discussion

The main features of the replication protocol are:

1. **Flexibility**: The flexibility of requesting locks at hot spots during the lock acquisition phase, pessimistically, (reduces the abort rates due to high conflicts) and requesting locks on other data items during the validation phase, optimistically, (provides higher transaction throughput), gives better performance. For any given transaction, a coordinator may decide to request a few of locks pessimistically and a few locks optimistically.

2. **Lower response time:** We know that a local transaction does not communicate with any other nodes to execute the primary transaction. If the transaction access pattern is known in advance, the custodianship for the data items can be arranged in such a way that global transactions can be minimized. Even though custodianship of the node is completely distributed in the registry, good design for assigning custodianship can give lower response time even with the increase in the number of nodes.

3. **Minimizing the bookkeeping required**: Book keeping for the session guarantee mechanism is minimized as it is proportional to the number of nodes instead of the number of data items. As a result, the protocol is highly scalable with an increase in data items in the registry.

4. **Distributed and fault tolerant**: As the latest updates may be at different nodes in the entire registry, the protocol designed is distributed. There is no single point of failure. If a node fails, the custodianship of data items in custody of that node is transferred to another node [UDDI].

5. **Fine grained session guarantee**: While ensuring an increasing view of the registry to a user, the protocol does not wait till exact previously known state is attained. The fine grained session guarantee waits only for the execution of previously read conflicting transactions with which the present transaction conflicts either directly or indirectly. This mechanism neither requires full synchronization nor tight coupling between the nodes.

6. **No deadlocks**: The fully optimistic replication protocol, which is an extension to the replication protocol, prevents the deadlock in the system by ensuring that there is no circular-wait among the transactions.

4.6 Performance evaluation

In this subsection, we analyze the performance of our replication protocol with respect to various parameters. The performance of the system can be increased by reducing each of the following costs associated with the execution of a transaction:

1. The communication cost: This is the overhead involved in sending and receiving the change-record messages from one node to another.

2. Execution cost: This is the cost involved in executing a transaction submitted at a node by its TM.

3. Waiting cost: The ROT has to wait at a node to attain a state such that session guarantee mechanism is satisfied. This waiting time is calculated to determine the throughput of the system.

Each of the following parameters affects the costs in different ways:

1. Number of data items in the registry.
2. Number of nodes in the registry.
3. Rate of conflicting transactions: This is determined by the number of conflicting transactions among the total number of transactions in the system.
4. Number of remote accesses of a transaction: To execute a transaction, the primary transaction coordinates with other nodes which are the custodians of the data items of its operations. The number of remote accesses is determined by the number of operations accessing data items in custody of other nodes.

In our study, we assume that custodianship of data items is uniformly distributed in the registry. This uniform distribution enables fair load balancing in the execution of the primary transactions in the registry.

Let us consider each of the costs with respect to above parameters.

1. Communication cost

As the number of operations of a transaction increases, the rate of conflict with other transactions increases. As the message propagation mechanism has to wait for the conflicting transactions, the wait time to deliver the change-record message at the destination node increases. As a result the communication cost increases. For example, when the number of operations of a transaction increases from five to ten, in the worse case the transaction conflicts with ten transactions instead of five. If all these conflicting transactions were executed at different nodes then the transaction has to wait for the change-record message from ten nodes instead of five. The additional wait time increases the communication cost of the system.

As the number of nodes in the registry increases, wait time to deliver a change-record of a transaction at a node increases. In the worse case, if the preceding primary transactions of a transaction are executed at different nodes, the change-record message delivered at the node is on hold until the preceding conflicting transactions are delivered. As number of nodes in the registry increases from two to five nodes, the wait time to deliver the change-record increases drastically. When number of

nodes is far more than the number of data items in the registry, increase in number of nodes does not have much affect on the wait time to deliver the change-record.

As the number of remote accesses of a transaction decreases, the conflicting transactions being executed at other nodes decreases. This decreases the wait time to deliver the transactions at a node.

2. Execution cost

As the number of nodes in the registry increases, the number of copies of a data item in the registry increases. Therefore, more number of ROTs can be executed in the registry at a given point in time. For example, when number of nodes in the registry is increased from five to ten nodes the ROTs can be executed at different nodes concurrently. This increases the transaction throughput of the system.

As the number of remote accesses of a transaction increases the communication overhead involved increases. This decreases the number of transactions executed per unit time.

3. Waiting cost

The session guarantee mechanism ensures that the ROT sees an increasing view of the system by waiting for all the preceding conflicting transactions in the session. As the number of transactions in a session increases, the session guarantee mechanism delays the execution of the ROT until the preceding conflicting transactions of the session are executed at that node. As a result, wait time to execute ROTs increases and throughput of the system decreases. For example, when the number of

transactions increases from five to ten, the tenth transaction in the worse case has to wait for previous nine transactions to execute, whereas the fifth transaction has to wait only for the previous four transactions to execute.

The increase in the number of data items in the registry does not effect wait time to execute a transaction in a session unless the data items induces a conflict with the other transactions in the session.

4.7 Starvation

As a few of the operations of a transaction may be executed optimistically, the transaction may be aborted several times. Thus, a few transactions may not be executed even after waiting for a long duration of time. This leads to starvation of those transactions.

For simplicity, first let us assume that all the transactions at a node are executed by the local TM using First Come First Serve (FCFS) policy. That is, a transaction which arrives first at a node is executed first by the local TM at that node. A simple and efficient solution to resolve starvation is to execute the transaction pessimistically by requesting locks on all the data items before the execution starts. Hence, a transaction is executed in the order of its arrival which ensures that the transaction is not aborted later due to conflicts.

The major problem with this method is that pessimistic locking increases the deadlock rate. The deadlock algorithm may select the same transaction as the victim transaction several times.

In the distributed system, an aborted transaction can be executed at a node which is not the same node as its previous execution. As a result an aborted transaction may be aborted again by a different TM. Therefore, it is not suitable to use FCFS method by local TM for resolving starvation at the global level. The following method is employed to resolve the starvation at a global level.

When a transaction is submitted to a local TM at a node for execution, the transaction is inserted in a FCFS queue at that node. When a transaction is scheduled for execution for the first time, it is assigned the NA-array at that node and initialized. Every transaction is assigned the NA-array as follows:

Let T_D, T_E, T_F be transactions with NA-arrays $NA_D[1,2,3,...,n]$, $NA_E[1,2,3,...,n]$ and $NA_F[1,2,3,...,n]$, respectively. NA-arrays for T_D, T_E, T_F are initialized at Node-X, Node-Y and Node-Z, respectively, as follows:

Then, $NA_D[1,2,3,\ldots,n] := N_X[x][1,2,3,\ldots.n]$

$NA_E[1,2,3,\ldots,n] := N_Y[y][1,2,3,\ldots.n]$

$NA_F[1,2,3,\ldots,n] := N_Z[z][1,2,3,\ldots.n]$

An aborted transaction retains the NA-array which was assigned to it in its previous execution. Upon addition of a transaction in the queue, it is sorted according to the precedence of NA-array. The precedence of the NA-array for transactions can be determined as follows:

if $NA_D[1,2,3,...,n] \leq NA_E[1,2,3,...,n] \leq NA_F[1,2,3,...,n]$

then their ordering in the queue is changed to $T_D \rightarrow T_E \rightarrow T_F$

This ordering is changed upon the addition of a new item into the queue. If ordering between two consecutive transactions cannot be determined, then they are unaltered. That is, its default FCFS ordering is maintained.

4.8 Livelocks

Livelock is a special case of deadlock in which the involved transactions constantly change their states with respect the states of others but do not make progress. It is also defined as a special case of resource starvation in which a specific transaction is not progressing.

In our protocol, a livelock occurs when two are more transactions are acquiring a few of the locks optimistically and other locks pessimistically. Both of the transaction types may be invalidated and aborted. During its next execution if those transactions swap the optimistic and pessimistic lock requests on their respective data items, this cycle may continue repeatedly. A simple method to resolve the livelock is to execute one transaction by requesting all the locks pessimistically at a point in time in the system. Only after the transaction finishes execution, another transaction is executed again in fully pessimistic method. In this way we can ensure that at least one transaction changes its state in the system.

4.9 Related Work

A brief comparison of our work with related works in the literature is as follows:

[HSAE03] uses the lazy replication where the primary transactions can be executed at any node. Once they have been successfully executed, they are said to be in precommit state. They are propagated to other nodes by *epidemic propagation* (exchanging up-to-date information by choosing another node at random, in a way passing them through the system like an infectious disease) of messages to detect conflicts. The transaction commits successfully if no conflicts are detected at any other node. Their method is in a way an eager replication where a pre-committed transaction commits only after it has the information of all the other nodes in the registry or it can abort if some other node does this computation and sends the abort message. Our method is more efficient than [HSAE03], as once a primary transaction at the coordinator commits, the user obtains the response from the system and coordination is required only with participant nodes. That is, our method needs to know only a subset of all the transactions in the system. While in [HSAE03], at all the nodes the data items in the write-set of the precommit transaction cannot be read by other transactions, whereas in our method the transactions can read values of previous transactions.

Providing session guarantees within a transactional framework is presented in [DS04]. They ensure a new correctness criterion called *strong session serializability* (strong session 1SR) which is weaker than strong serializability (where all conflicting

transactions must be serialized in the order in which they are submitted) but stronger than 1SR. Our method is more efficient than in [DS04], as we provide the flexibility of executing ROTs at any node in the registry, whereas they allow only at specified nodes. As the session keeps track of transactions from which it has read, instead of the present state of the registry our session guarantee mechanism is fine grained and provides more concurrency than theirs. We consider the conflicts (directly or indirectly) only on those data items which the present transaction is accessing, whereas they consider the exact state of the system for comparison. We allow distributed transaction execution, whereas theirs is centralized.

Recently, providing the freshness guarantee in partially replicated databases has been dealt with in the PDBREP project in [ATGB05]. It considers a lazy replicated database where there are separate set of update nodes and read only nodes. They assume that system ensures 1SR criterion at update nodes. As any read only node is not as up-to-date as an update node, they allow user to specify the freshness requirements for transactions. PDBREP has a centralized log which keeps track of the present state of the system, whereas ours is completely distributed. In their case, if user gives an invalid freshness requirement, the user may read an inconsistent view of the system. In our protocol, we first provide 1SR. Then, we provide Monotonic Read Only and Read Only Follow Updates guarantees which ensure the valid and consistently increasing view of system to each user in the session individually.

In traditional name service directories, such as Grapevine [S84] and Clearinghouse [X84], a weak consistency called eventual consistency is employed. They are not serializable but all data items of all the nodes eventually reach the same state. It is a sufficient correctness criterion for the applications for which they are used. Due to the lack of trust between organizations and other issues, such as confidentiality, in UDDI a serialized view of transactions is required. Any weaker consistency criterion should be avoided. In [SLK04], a comparison of UDDI registry replication strategies is given.

[CRR96] considers replication in partially replicated databases. A data item can be in custody of any one of the nodes in the system. As they consider the basic reliable broadcast, there is a need to ensure global serializability. They consider the issue of assigning primary node for data items. They use the data placement graph which is a directed graph representing distribution of primary and secondary copies of data items across the system. They show that acyclic data placement graph ensures global serializability. Their algorithm finds a primary node for each replicated data item. The main drawback of their algorithm is that, it finds the primary node for a data item only when such a solution exists. It provides only very restricted set of solutions. When there is no feasible solution, the algorithm does not take any alternative action. It requires a lot of careful designing or the system would not ensure transactional guarantees. Our protocol is more efficient than theirs, as whatever may be the

distribution of the custodianship in the replicated registry; the transactions can be executed (provided there are no deadlocks and failures).

Chapter 5

Partial Replication

Replicating all the data items at all the nodes requires high bandwidth and many of the replicated data items may never be used. Replicating only those data items which are frequently used is a practical approach to increase the performance and utility of the system. Replicating a data item only at a subset of the nodes is called *partial replication.* If the data item's access pattern is known in advance, then the replicated system can be designed such that data items are only replicated at the nodes where they are used. Also, more frequently used data items can be replicated at more nodes than less frequently used data items, thereby, increasing the availability and utility of the system. The partial replication reduces lock contention, write overhead and communication costs in comparison with the fully replicated system. This aspect improves the performance of the distributed system. Ideally, partially replicating required data items in the system should increase the system performance in comparison with the fully replicated system. In reality, the advantage of partial replication is offset by the cost of added complexity that is required to manage it, in comparison to full replication.

In a partially replicated system, a data item is replicated only at a few nodes. If a transaction updates a data item, only those nodes in the system having a copy of the data item need updates of the transaction. Instead of broadcasting updates of the

transaction to all the nodes in the system, it is advantageous to multicast these updates to only the subset of the nodes which have these data items. Also, at the destination node where the message is to be delivered, it does not need to wait even for a preceding conflicting transaction in the system that does not update any data item at that node. Similarly, for the session guarantee mechanism for a ROT executing at a node, the ROT does not need to wait for a change-record which is not sent to that node. Our motivation for using the causal multicast mechanism is demonstrated by the following example.

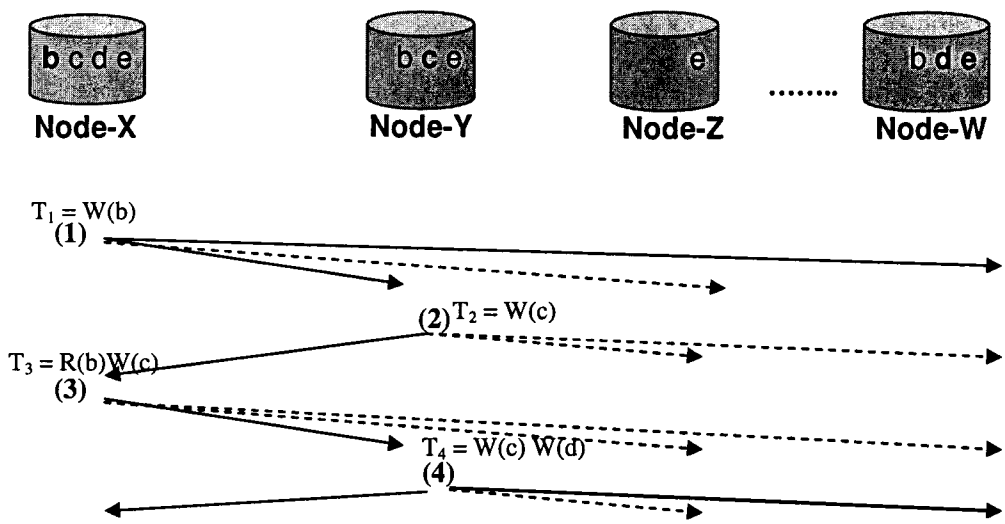

Figure 5.1 Illustrates the benefits of causal multicast over causal broadcast

Example 5.1: Consider the setup as shown in Figure 5.1. Let us consider a registry with 100 nodes, say Node-X, Node-Y, Node-Z, Node-Z1, Node-Z2,..., Node-W.

Data item b is replicated at Node-X (custodian), Node-Y, and Node-W. Similarly, data item c is replicated at Node-X and Node-Y (custodian). Data item d is replicated at Node-X and Node-W (custodian). Data item e is replicated at all the nodes (Node-W is the custodian).

A simple solution to ensure global serializability of such a partially replicated system would be to use the causal broadcast mechanism (as discussed in chapter 4). The change-record can be ignored at the nodes where there are no data items to update.

T_1, T_2, T_3, and T_4 are transactions as shown in the Figure 5.1. Their respective change-records are indicated by (1), (2), (3), and (4). The solid lines indicate nodes at which change-records, once delivered, are implemented as refresh transactions because these nodes contain data items in the write-set of the transaction. The dotted lines indicate the nodes where the delivered change-records are ignored. In summary,

- (1) is implemented at Node-X, Node-Y, and Node-W.
- (2), (3) are implemented at Node-X and Node-Y.
- (4) is implemented at Node-X, Node-Y, and Node-W.

Let us assume that the communication link between Node-X and Node-W is broken after (1) is delivered at all the nodes. Then, the causal broadcast mechanism delivers (4) at Node-W only when the link starts working and all the preceding change-records, namely, (2) and (3), have been delivered at that node. (4) is waiting for other change-records (i.e., (2) and (3)), which will eventually be ignored at Node-W. In

summary, at Node-W, message (4) waiting for other change-record messages, namely, (2) and (3), is an unnecessary wait. On the other hand, if the causal multicast mechanism is used, then (4) can be delivered without waiting for any other messages. This method, in turn, increases the transaction throughput of the system.

In partially replicated system, all the data items accessed by a transaction may not be at a single node. This factor affects the update transactions and ROTs as follows:

- Usually, a node which has copies of all the data items accessed by a transaction is selected as the coordinator for the execution. If a transaction accesses a data item which does not exist at the coordinator node, then the coordinator can request the lock and value from the custodian node of that data item.
- In a replicated system, the volume of ROTs is usually high. To increase the transaction throughput of the system, we provide access to any value of a data item as permitted by the session guarantee mechanism. In partially replicated systems, as all the data items are not present at a particular node, we extend a ROT to access multiple nodes. A ROT which accesses data items at more than one node in the registry is called the global ROT. As we have global ROTs in the system, even if the TM at each node in the registry ensures serializability, it may not be possible to ensure serializability (1SR) at the global level. This can be illustrated with the following example.

Example 5.2: Consider the setup shown in Figure 5.2. Let Node-Z be the custodian of data items b and c. Let Node-X and Node-Y have data items b and c, respectively. We have the following transactions in the system:

$T_K \rightarrow W(b)\ R(c)$

$T_L \rightarrow R(c)\ W(c)$

$T_H \rightarrow R(b)\ R(c)$

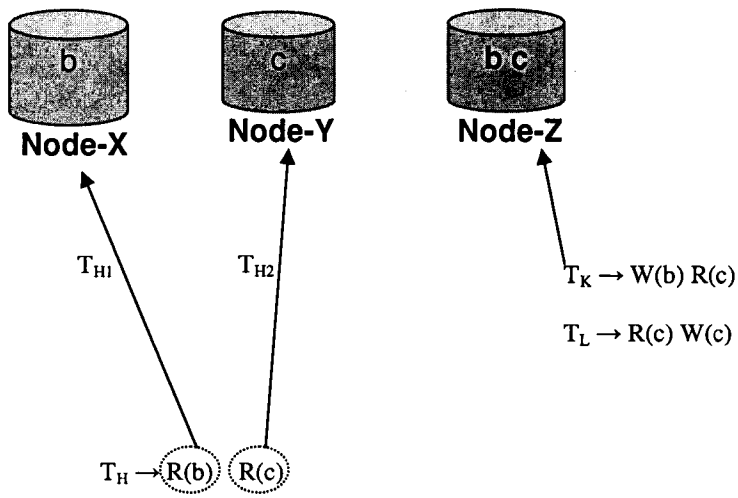

Figure 5.2 Illustrates the inconsistent state seen by the global ROT, T_H

They are executed as follows:

a) T_K and T_L are executed at Node-Z. T_K precedes T_L ($T_K \rightarrow T_L$).

b) T_H is divided into sub-transactions T_{HI} and T_{H2}. T_{HI} reads data item b at Node-X, before T_K is executed ($T_{HI} \rightarrow T_K$) and T_{H2} reads data item c at Node-Y, after T_L

is executed ($T_L \rightarrow T_{H2}$). As T_H should be executed atomically as a single transaction, the execution order is $T_L \rightarrow T_H \rightarrow T_K$.

From (a) and (b) we have a cycle, $T_L \rightarrow T_H \rightarrow T_K \rightarrow T_L$ in the serialization graph at a global level. Because an atomic global transaction is divided into two sub-transactions, the schedule is not globally serializable. This is an incorrect execution. (Although, in this thesis, we consider two different ROTs from two different sessions creating a cycle in the global serialization graph to be a correct execution.)

- Also, if the same message propagation mechanism (i.e., broadcasting), as in the fully replicated system, is used for multicasting; it does not ensure the liveness property. This can be illustrated with the following example.

Example 5.3: Consider the setup as shown in Figure 5.3. Node-X and Node-Z are custodians of data items b and d, respectively. T_L and T_H are transactions executed at Node-X and Node-Z, respectively. For any update transaction in the system, its change-records are multicast to nodes containing data items in the write-set of the transaction, as only those nodes needs to update the data item. Consider the following execution sequence:

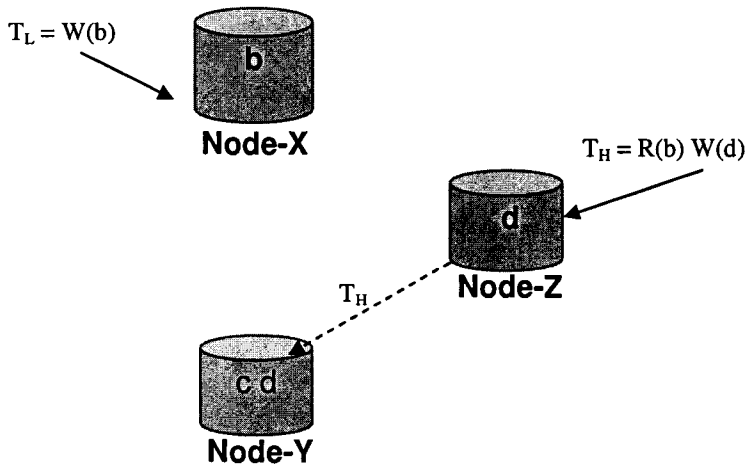

Figure 5.3 Illustrates that the liveness property is not ensured by the causal multicast mechanism

- T_L is executed with Node-X as the coordinator.
- T_H is executed with Node-Z as the coordinator. Node-Z requests the lock and value for data item b from Node-X. After obtaining the lock and value of data item b, T_H executes R(b). T_H commits and the change-record of T_H is sent to Node-Y. As T_H obtains lock on data item b after the execution of T_L at Node-X, T_L precedes T_H, i.e., $T_L \rightarrow T_H$.
- At Node-Y, the change-record of T_H is waiting for the change-record of T_L, as it is a preceding conflicting transaction. T_L will never be delivered at that node, as it was not sent to Node-Y by its coordinator. (Please note that the change-record of T_L is not sent to Node-Y, as there is no data item b at that node.) Therefore, T_H will be waiting for T_L infinitely.

Therefore, the liveness property for the delivery of messages is not ensured by the causal message propagation mechanism.

The message propagation mechanism employed in the fully replicated system must be modified to ensure the liveness property. We know that serializability at the global level is dependent on the causal propagation mechanism. While designing the propagation of messages, we have to ensure that, on one hand, there is progress in the message delivery and, on the other hand, the dependent messages are delivered in the same serial order. Therefore, to deliver a change-record of a transaction at a node, the multicast mechanism has to know the transaction's preceding conflicting transactions and which of these conflicting transactions have been sent to that node by their coordinators.

5.1 Protocol for partial replication

Analogous to the fully replicated protocol, the execution of a transaction is divided into a primary and a refresh transaction. The protocol is explained from the coordinator's view, participant's view, and non-participant's view.

The protocol for partial replication differs from that of full replication in the following ways:

1. Message propagation: Implements the causal multicast of messages, instead of the causal broadcast of messages.
2. Replication protocol: Replicates a data item at only a subset of nodes, instead of replicating at all the nodes.

3. Session guarantee mechanism for read only transactions: Implements global ROTs.

A change-record for T_K contains the following fields:

1. Node identifier of the coordinator of T_K;
2. USN of transaction T_K;
3. Write operations with their values and read operations of transaction T_K;
4. Present state array ($P_Y[1,2,3,\ldots,n]$) of the coordinator (optional);
5. Dependency array (D-array) of T_K. $D_K[1,2,3,\ldots,n][1,2,3,\ldots,n]$ of T_K which contains the USN of T_K and USNs of all the preceding (directly and indirectly) conflicting transactions; and
6. Receive array (R-array) of T_K: As the change-record of a transaction may not be sent to all the nodes in the partially replicated protocol, the liveness property cannot be ensured using the protocol in chapter 4 (see Example 5.3). To solve the problem cited in the example, an additional array called the R-array is used to indicate that T_H does not have to wait for T_L at Node-Y, as T_L was not sent to Node-Y by its coordinator. $R_H[y][1,2,3,\ldots,z-1,z+1,\ldots,n]$ of T_H indicates the USNs of all the preceding (directly and indirectly) conflicting transactions which have to be delivered at Node-Y, before T_H can be delivered at that node. Hence, $R_H[y][1,2,3,\ldots,z-1,z+1,\ldots,n]$ is designed in such a way that its change-record waits for only those preceding conflicting transactions which were sent to Node-Y

by their coordinators. In summary, the R-array of a change-record is designed in such a way that to deliver the change-record at a node, it waits only for those preceding conflicting transactions which were sent to that node by their coordinators.

D-arrays and R-arrays are associated with the change-record, lock-grant, and acknowledgement messages.

5.1.1 Primary transaction

For Coordinator (Coordinator's view)

Consider the execution of the primary transaction, T_K, at the coordinator, Node-Y.

(1) Lock acquisition phase: T_K acquires the locks in two steps, namely, stage (1.a) and stage (1.b), executed in the given order. Once all the requested locks have been acquired, the protocol goes to stage (2).

(a) Requests PT-S and PT-X locks on data items which are in custody of Node-Y, atomically. After successful completion the protocol goes to stage (1.b).

(b) The coordinator of T_K decides which locks to request from the participants now (in stage (1.b)). If there is a read operation in T_K accessing a data item not present at the coordinator node, then the lock and value for that data item must be requested from the participant (now). That is, transaction T_K cannot execute that operation optimistically. A PT-S lock-value-request message on that data item is sent to its custodian node by the coordinator. On the other

hand, if a data item is available locally, a PT-S lock-request message on the data item is sent to its custodian node by the coordinator.

(2) Execution phase: Transaction T_K is executed. All writes are performed in the private workspace.

(3) Validation phase: Let us consider the validation of the operation on data item b of transaction T_K executed at the coordinator, Node-Y. If a transaction is performing validation of an operation on a data item, it means that it had not requested the lock on that data item in stage (1). If both read and write operations are performed on the same data item optimistically, then the write operation on that data item should precede the read operation. The procedure for validation of an operation on a data item of a read operation only or a write operation only is the same as in the fully replicated protocol. The types of validation messages that are possible in the partially replicated protocol are:

(a) Read validate a data item: Possible only if the data item is present at the coordinator of T_K.

(b) Read and write validate a data item: Possible only if the data item is present at the coordinator of T_K, or if the data item is not present at the coordinator but the write operation precedes the read operation.

(c) Write validate a data item: Possible.

S-optimistic-request and X-optimistic-request messages are sent to read validate and write validate, respectively. While sending the S-optimistic-request message

on data item b, the identifier of the transaction, <Node-ID, USN>, from which the data item was read is also sent.

(4) Abort phase: The protocol aborts the transaction, as per the decision made in the validation phase. Then, it sends the lock-release message to all the participants of the transaction. This procedure is the same as in the fully replicated protocol.

(5) Commit phase: The transaction commits by executing the following steps in the given order:

a) Executes the refresh transaction protocol (explained later) of T_K.

b) Commits the primary transaction of T_K.

c) Creates the change-record of T_K. First, the USN is generated. Then, updates the state array N_Y, creates P-array P_Y, D-array D_K, and R-array R_K.

d) Releases all the PT locks.

e) Multicasts the change-records to all other nodes which have at least one data item in the write-set or the read-set of T_K.

Then, the *session_update* procedure is executed. This procedure is discussed later.

When the registry starts, the state vector at each node is initialized to zero value as follows:

$N_X[1,2,3,\ldots,n][1,2,3,\ldots,n] := N_Y[1,2,3,\ldots,n][1,2,3,\ldots,n] := 0$
$N_Z[1,2,3,\ldots,n][1,2,3,\ldots,n] := N_W[1,2,3,\ldots,n][1,2,3,\ldots,n] := 0$

The procedure for creation and updating of arrays in stage (5.c) can be explained as follows:

- Update of $N_Y[1,2,3,\ldots,n][1,2,3,\ldots,n]$ at Node-Y

$N_Y[x][z]$ is the USN of a transaction whose primary transaction was executed at Node-Z, and has been executed at Node-X as refresh transaction that Node-Y is aware of. (Sometimes, $N_Y[x][z]$ may indicate the USN of a transaction which was not executed at Node-X. This happens when Node-X is the custodian of the data item in the read-set of the transaction, but does not contain any data item in the write-set of the transaction.) The state array at Node-Y is updated so that it records the commitment of T_K at that node.

$N_Y[y][y]$:= USN of primary transaction T_K

- Creation of the present state array (P-array), P_Y, representing the y^{th} row of N_Y

P-array is a one dimensional array indicating the state of the coordinator. This array is optionally sent with the change-record.

$P_Y[1,2,3,\ldots,n] := N_Y[y][1,2,3,\ldots,n]$

- Creation of the R-array, R_K, of T_K

$R_K[x][z]$ is the USN of either the transaction conflicting with T_K, or of T_K, whose primary transaction is executed at Node-Z (its coordinator) and has been sent to Node-X. In the Z^{th} column of the R-array, rows that have the USN of T_K indicate the nodes to which the change-records of T_K have been sent. These nodes include the coordinator, participants, and those non-participants which have the data items in the

write-set of the transaction. The procedure below stores the USN of T_K in the rows of R-array, corresponding to nodes that are sender and receivers of the message of T_K.

$R_K[1,2,3,\ldots,n][1,2,3,\ldots,n] := 0$ // initialize
$R_K[y][y]$:= USN of transaction T_K
For all x, to which the change-record of T_K is sent, $R_K[x][y]$:= USN of transaction T_K
Here all nodes indicated by x are participant nodes or those non-participant nodes which have a data item in the write-set of the transaction.

A row in the R-array of T_K indicates the preceding conflicting transactions, which have to be delivered at a node represented by the row in order to deliver the change-record of T_K at that node. For example, $R_K[x][1,2,3,\ldots,n]$ of T_K indicates to Node-X, the preceding conflicting transactions that have to be delivered at that node before delivering that transaction.

- Creation of D-array D_K for T_K

$D_K[x][z]$ is the USN of a transaction conflicting with T_K, or of T_K, whose primary transaction was executed at Node-Z (its coordinator) and Node-X as the participant. In the Z^{th} column of the D-array, the rows with the USN of T_K indicate the coordinator and participants of T_K. The procedure below stores the USN of T_K in the rows of the D-array, corresponding to nodes that are participants and coordinator of T_K.

$D_K[1,2,3,\ldots,n][1,2,3,\ldots,n] := 0$ // initialize
$D_K[y][y]$:= USN of transaction T_K
For all x, which are participant nodes of T_K, $D_K[x][y]$:= USN of transaction T_K.

Let T_K conflict with a set of transactions T_{K1}, T_{K2},...,T_{KM} at the present node. Let D-arrays and R-arrays corresponding to these conflicting transactions be D_{K1}, D_{K2},...,D_{KM} and R_{K1}, R_{K2},...,R_{KM}, respectively. (Please note that the D-array and the R-array can be obtained from the change-records of these transactions.) Also, let the D-array and the R-array associated with the lock-grant and acknowledgement messages of T_K be D_{KM1}, D_{KM2},...,D_{KP} and R_{KM1}, R_{KM2},...,R_{KP}, respectively. Then, the procedure below computes the transactions that are conflicting with T_K, and the nodes to which these transactions have been sent to and stores them in arrays.

for var = 1 to n, where n is the number of nodes
  for varp = 1 to n, where n is the number of nodes
    R_K[var][varp] := Max(R_{K1}[var][varp], R_{K2}[var][varp],...., R_{KM}[var][varp], R_{KM1}[var][varp], R_{KM2}[var][varp],...,R_{KP}[var][varp], R_K[var][varp])
    D_K[var][varp] := Max(D_{K1}[var][varp], D_{K2}[var][varp],...., D_{KM}[var][varp], D_{KM1}[var][varp], D_{KM2}[var][varp],...,D_{KP}[var][varp], D_K[var][varp])

The above code calculates and stores in D_K, USNs of all the transactions conflicting with T_K in the entire system. Similarly, USNs stored in R_K indicates the nodes to which the change-record of these conflicting transactions has been sent.

In Figure 5.4, transaction T_H is executed with Node-Y as the coordinator. Node-Y sends a PT-S lock-value-request on data item b to Node-X in stage (1.a). After obtaining the lock-value-grant, the protocol executes the transaction by performing the operations on data item d optimistically. Later, the coordinator of T_H validates those read and write operations on data item d, by sending the validation request message to Node-Z. Upon successful validation, transaction T_H commits.

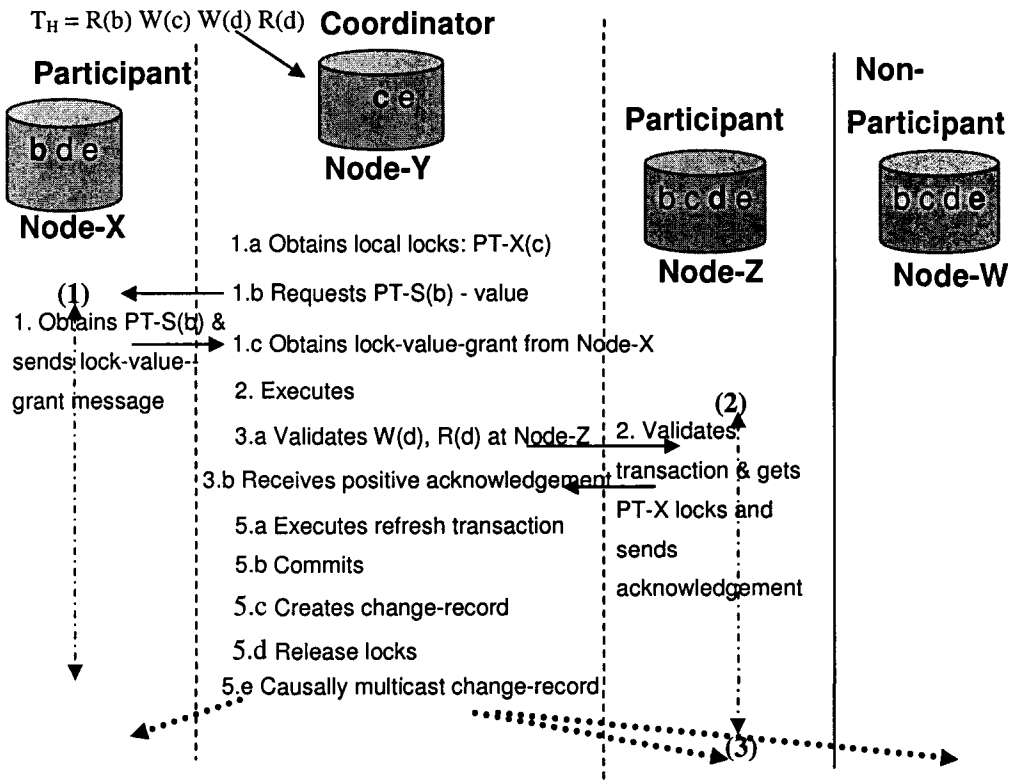

Figure 5.4 Illustrates the execution of the primary transaction protocol of T_H

For Participants (Participant's view):

Consider a participant node, Node-X, participating in the execution of the primary transaction of T_K:

(1) Upon receiving a PT-S/PT-X lock-request or a PT-S lock-value-request (refer to stage (1.b) of the coordinator's view): For a PT-S (PT-X) lock-request message, Node-X acquires a PT-S (PT-X) lock on the data item from the local TM. Then, it sends the lock-grant message along with the D-array, $D_{KRI}[1,2,3,\ldots,n][1,2,3,\ldots,n]$,

and the R-array, $R_{KRI}[1,2,3,\ldots,n][1,2,3,\ldots,n]$ (That is, Node-X is the i^{th} participant of T_K). For a PT-S lock-value-request on a data item, Node-X acquires the lock on the data item and sends a lock-value-grant message, along with the present value of the data item, the D-array, $D_{KRI}[1,2,3,\ldots,n][1,2,3,\ldots,n]$, and the R-array, $R_{KRI}[1,2,3,\ldots,n][1,2,3,\ldots,n]$. These message types may optionally contain the P-array.

(2) Upon receiving the validation message (refer to stage (3) of the coordinator's view): A participant can receive an S-optimistic-request message or an X-optimistic-request on a data item. Validation of an operation on a data item is performed as follows:

a) Upon receiving the S-optimistic-request: The S-optimistic-request message contains the identifier of the transaction <Node-ID, USN> from which the data item b was read. A check is made if there are any writes on data item b starting from the latest state of the node, until the change-record identified by <Node-ID, USN>, or the last change-record is reached. If true, a negative acknowledgement is sent. Otherwise, a PT-S lock is obtained from the local TM (We assume that checking the condition and obtaining the locks are performed atomically) and a positive acknowledgement is sent.

b) On receiving the X-optimistic-request: For an X-optimistic-request on data item d, Node-X obtains a PT-X lock from the local TM and sends a positive acknowledgment. If the PT-S lock is already present on the data item by the

same transaction, then the lock is upgraded to the PT-X lock and a positive acknowledgement is sent. Please note that in-order to upgrade a lock on a data item, the TM at Node-X must wait until all the other conflicting transactions holding read locks on that data item are released. Deadlocks are possible when two or more transactions are trying to upgrade their locks on the same data item simultaneously.

c) Both S-optimistic-request and X-optimistic-request: On receiving both the requests on data item c, first, a check is performed as in the case of the S-optimistic-request. Then, upon a successful outcome, the PT-X lock on the data item d is acquired and a positive acknowledgement is sent.

All the acknowledgement messages contain the D-array and the R-array. They may optionally contain the P-array.

(3) Upon delivery of a change-record (refer to stage (5.e) of the coordinator's view): The change-record is stored. Then, one of the following steps is executed.

a) If there are data items of the write-set of the transaction at Node-X, then the refresh transaction protocol is executed. Please note that the refresh transaction writes only the data items in the write-set of the transaction present at that node. The other operations in the write set of the transaction are ignored.

b) On the other hand, if data items of the write-set of the transaction are not present at Node-X, then the refresh transaction protocol is not executed.

Later, the participant node releases all PT locks of transaction T_K.

(4) Upon receiving the lock-release message (refer to stage (4) of the coordinator's view): Releases all the PT locks held by transaction T_K.

For Non-Participants (Non-Participant's view):

Upon receiving the change-record from the coordinator node, it is stored and the refresh transaction protocol is executed.

5.1.2 Refresh transaction

For Coordinator and Participants (Coordinator's view and Participant's view):

Consider the execution of the refresh transaction of T_K at Node-X.

1) Lock acquisition phase: The protocol request for RT locks and waits until conflicting locks are released. Then, RT-X locks are acquired on all the data items to be written at that node. A RT-X lock on a data item can be given even when the conflicting transaction has a PT-X lock on that data item. (Please note that as RT-X and ROT-S locks are incompatible, a RT-X lock on a data item is not given if there is any ROT-S lock on that data item.)
2) Execution phase: Transaction T_K is executed. Now, data items of T_K's write-set present at that node are written.

3) Commit phase: PT-S, PT-X, and RT-X locks on all the data items of T_K are released atomically.

For Non-Participants (Non-Participant's view):

The procedure is the same as in the protocol for the fully replicated case, except that the write-set of T_K present at that node is written.

5.1.3 Read only transaction

The protocol is the same as in the fully replicated protocol, except that in the session guarantee mechanism for the global and local ROTs.

5.1.4 Mechanism for session guarantee

We need an additional array called SR_P for the session guarantee mechanism. This array serves to eliminate the idle waiting at a node for the change-record of a transaction which is not sent at that node. This can be illustrated using the following example.

Example 5.4: Consider the set up shown in Figure 5.5. T_L and T_H are executed at Node-X and Node-Z, respectively, where T_L precedes T_H ($T_L \rightarrow T_H$). Let us assume that a new session starts with a transaction reading data item d written by T_H at Node-Z. When the next transaction of the session with a read operation on data item c is executed at Node-Y, it must wait for the change-record of T_H and T_L, as they are conflicting transactions. As the change-record of T_L is not sent to Node-Y, the ROT infinitely waits for the change-record of T_L to be delivered. A better solution would

be to allow the ROT to read the values from Node-Y, without delivery of T_L at that node, as it would still provide the consistent view of the system.

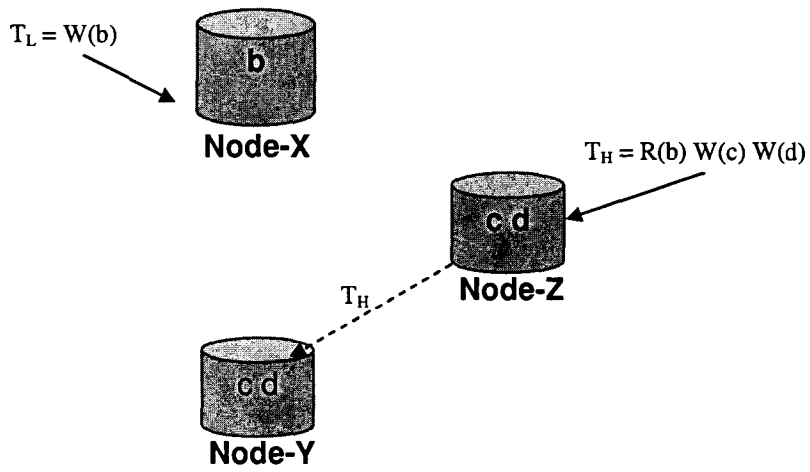

Figure 5.5 Illustrates that the liveness property is not ensured by the session guarantee mechanism

Initialization:

The data structures used in the session guarantee mechanism are initialized as follows:

$S_P[1,2,3,...,n][1,2,3,...,n] := 0$ // belongs to a user session
$SR_P[1,2,3,...,n][1,2,3,...,n] := 0$ // belongs to a user session
$M_X[1,2,3,...,n][1,2,3,...,n] := M_Y[1,2,3,...,n][1,2,3,...,n] := 0$ // stored at a node
$M_Z[1,2,3,...,n][1,2,3,...,n] := M_W[1,2,3,...,n][1,2,3,...,n] := 0$ // stored at a node

Session_read:

This procedure is performed before the execution of a ROT. Its key features are:

- The procedure eliminates the idle waiting for change-records which were not sent to the node where the ROT is being executed.

- It provides the flexibility for a global ROT to read data items from more than one node. This is performed by dividing a global ROT to a number of sub-transactions and executing them at those nodes.

The global ROTs are useful for the following purposes:

- An administrator of the registry may want to archive the status of the registry at the end of the day for auditing purposes. As only the custodian nodes of the respective data items have the latest updates, no single node may have an accurate and current snapshot of the entire system. A global ROT is divided into a number of sub-transactions and executed at their respective custodian nodes, as a ROT. In Figure 5.6, T_L is divided into sub transactions T_{L1}, T_{L2}, and T_{L3}. These transactions read data items b, c, and d at Node-X, Node-Y, and Node-Z, respectively. T_L obtains the latest state of the registry.

- As the number of customers in the registry increases, there may be different classes of users, namely, paid and free users. A business strategy could be to provide more recent updates to paid users and any values to free users. The free user's ROTs can be executed in such a way that they access data items only at the nodes that are not custodians of those items. In this way, any update transaction can obtain the response from the system without being interrupted by a free user's

ROT. This is because at the coordinator node of an update transaction, its primary and refresh transactions do not wait for the locks held by free user's ROT. In Figure 5.6, T_H is divided into sub-transactions T_{H1} and T_{H2}. These sub-transactions read local copies at Node-Y and Node-Z, respectively. Please note that if T_U and T_H are executed at the same point in time, one does not have to wait for another even though they are conflicting transactions. This method enhances the availability of the system.

The global ROTs may be used in the fully replicated systems also. In partially replicated systems, global ROTs become a necessity sometimes (depending on the data items the transaction accesses), as the data distribution is not uniform.

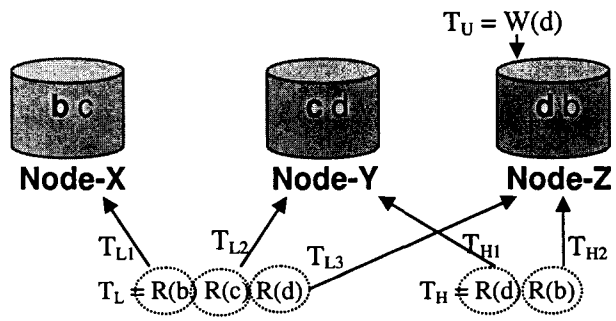

Figure 5.6 Illustrates the execution of global ROTs

For execution of the global ROT, a consistent global view of the system must be explicitly provided, as nodes being accessed by the sub-transactions may be at different states of the system. Please recall that in Example 5.2, T_H will obtain an

inconsistent global view of the system due to the same reason. The global transaction does not obtain a consistent view of the system, as different sub-transactions read different states of the system at different nodes. That is, in Example 5.2, T_{H1} at Node-X obtains the state of the system which was present before the execution of T_K, where as T_{H2} at Node-Y obtains the state of the system after the execution of T_L.

The difference between local and global transactions is that in the former case, a check is only performed before the execution of the transaction to ensure that the session reads an increasing state of the system. In the latter case, in addition to this check for component sub-transactions of the global transaction, it checks if a consistent state of the entire system is obtained by the global transaction.

The procedure to check if a ROT can be executed, *session_read_check*, is as follows:

S_P represents user session P, containing all the preceding (directly or indirectly) conflicting transactions seen by the user in the system. $SR_P[x][1,2,3,\ldots,n]$ represents all those preceding conflicting transactions of user session P which have been sent to Node-X by the transaction's coordinator node. Let us consider the execution of a ROT, T_K, at Node-X. T_K accesses the data items in custody of Node-Q, Node-R, and Node-W. Temp is a one dimensional array. The *session_read_check* procedure calculates the USNs of conflicting transactions of the ROTs that have been sent to that node. Then, the procedure waits only for those transactions that are to be executed at the node.

```
for var = 1 to n, where n is the number of nodes
  //check for change-records which should have arrived at each of the nodes
  tempR[var] := Max(SRP[q][var], SRP[r][var], SRP[w][var])
  where q, r, and w are custodian nodes of data items accessed by the ROT

wait until (NX[x][1,2,3,...,n] ≥ tempR[1,2,3,...,n])
 then execute the ROT at Node-X
```

In the above code, temp-array contains the USNs of the transactions for which the ROT has to wait. At Node-X, N-array is compared to check if these transactions have been executed already. If the change-record of a particular transaction has not been sent to Node-X, then it does not wait for that transaction.

Execution of the global ROT:

Let us consider the execution of a global ROT. The procedure can be explained as follows:

- Divide the global transaction into two or more sub-transactions. Submit the sub-transactions to different nodes. One of these nodes is selected as the coordinator. All the others are regarded as participants.
- *Session_read_check* procedure is executed at each of the nodes corresponding to the sub-transactions at that node.
- Participants send the result and other information, such as the D-array, N-array, and P-array, for each of those sub-transactions to the coordinator.

- The coordinator validates the global transaction using the *session_read_validate* procedure.

Let us consider the global ROT, T_G, which is divided into sub-transactions T_{G1}, T_{G2}, $T_{G3,...,}$ T_{GN} and are executed at different nodes. One of the nodes, for example Node-Y, is selected as the coordinator. Node-X is one of the participant nodes.

Session_read_validate procedure for the global transaction can be executed from the view of the coordinator and participant.

View of the coordinator node, Node-Y:

Consider the sub-transaction, T_{G2}, of the global transaction, T_G, executing at Node-Y.

1. Executes *session_read_check* procedure: This procedure is executed for sub-transaction T_{G2}. It obtains locks on data items and executes the sub-transaction.

2. Create D-array and R-array: This step finds the preceding conflicting transactions on all the data items that the sub-transaction, T_{G2}, accesses. The procedure for creating the D-array and R-array for the sub-transaction, T_{G2}, at Node-Y is as follows:

   Let us assume that T_{G2} conflicts with T_1, T_2, T_3,...,T_M. Let D-arrays and R-arrays of these transactions be D_1, D_2, D_3,...,D_M and R_1, R_2, R_3,...,R_M, respectively. Then, the USN of transactions conflicting with T_K and the nodes to which these conflicting transactions have been sent is given by the D-array and the R-array, respectively.

$D_{G2}[1,2,3,\ldots,n] = 0$
$R_{G2}[1,2,3,\ldots,n][1,2,3,\ldots,n] = 0$
for var = 1 to n, where n is the number of nodes
 for varp = 1 to n, where n is the number of nodes
 $D_{G2}[var]$:= Max($D_1[varp][var]$, $D_2[varp][var]$, $D_3[varp][var]$,...., $D_M[varp][var]$, $D_{G2}[var]$) // D_{G2} is one dimensional array
 $R_{G2}[varp][var]$:= Max($R_1[varp][var]$, $R_2[varp][var]$, $R_3[varp][var]$,...., $R_M[varp][var]$, $R_{G2}[varp][var]$) // R_{G2} is a two dimensional array

The above code computes and stores in arrays, D_{G2} and R_{G2} all the preceding conflicting transactions of T_{G2} and the nodes to which the change-records of these transactions are sent, respectively.

3. Validate the global transaction: The D-array, R-array, and P-array from all the participants of the global transactions are received. For a given P-array, the procedure checks with every other sub-transactions to ensure that the preceding conflicting transactions of those sub-transactions are executed at that node. This procedure can be explained as follows:

 After receiving D-array and R-array from Node-X, the coordinator, Node-Y, evaluates the acknowledgements as follows:

if ($D_{G2}[1,2,3,\ldots,n] \geq N_Y[y][1,2,3,\ldots,n]$)
 for z =1 to n, where n is the number of nodes
 if ($R_{G2}[y][z] \geq N_Y[y][z]$)
 then result is negative & exit
 end of for
else
 result is positive

In the above code, a comparison is performed between the present state of Node-Y and the D-array of the sub-transaction, T_{G2}, which was executed at Node-X. A positive result indicates that the state of Node-X and Node-Y are consistent with respect to the sub-transaction, T_{G2}.

Using the method explained above, the D-array of the sub-transaction of T_{G2} is compared with the present state of every other node at which other sub-transactions were executed.

Then, similarly to all the other sub-transactions of the global transaction, its D-array is compared with the present state array of every other nodes at which a sub-transaction of the global transaction is executed. If any one of the results is negative, then T_G is invalidated. Otherwise, after obtaining the positive results for all the comparisons, T_G is validated successfully.

4. Release locks: Releases all local locks and sends a lock-release message to all the participants. If validation is successful, then the results are returned to the user.

View of participant node, Node-X:

Consider the execution of the sub-transaction, T_{G1}, at the participant node, Node-X.

1. Execute *session_read_check*: This procedure is executed for sub-transaction T_{G1}. It obtains locks at Node-X and executes T_{G1}.
2. Create D-array, R-array, and P-array: This step finds preceding conflicting transactions on all the data items sub-transaction T_{G1} (executing at Node-X)

accesses, except for, those data items of which it is the custodian. It uses a similar procedure used for the coordinator to generate D_{G1} and R_{G1}. Then, it sends P-array, D-array, and R-array, i.e., $P_X[1,2,3,\ldots,n]$, $D_{G1}[1,2,3,\ldots,n]$, and $R_{G1}[1,2,3,\ldots,n]$, to the coordinator.

3. Upon receiving the lock-release request: All the locks of the sub-transaction are released.

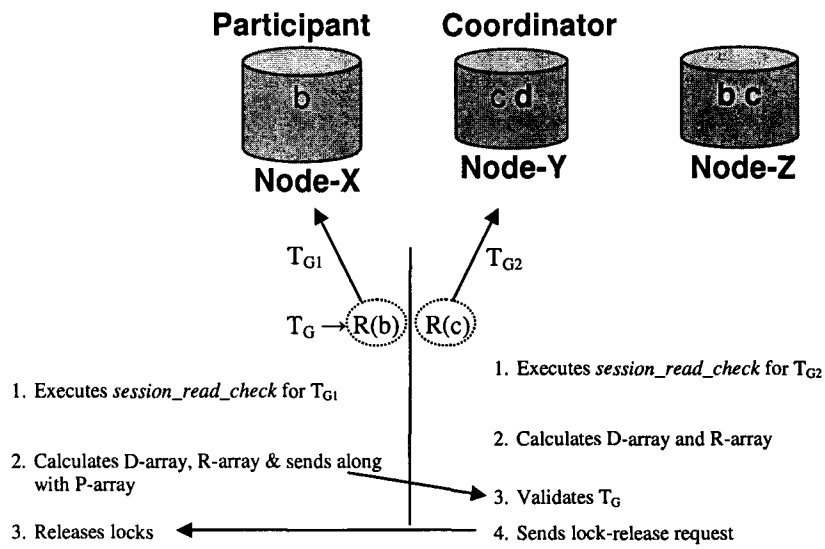

Figure 5.7 Illustrates the protocol for the execution of the global ROT, T_G

In Figure 5.7, T_{G1} and T_{G2} execute the *session_read_check* procedure at Node-X and Node-Y, respectively, where all the operations of the sub-transactions are executed. The coordinator, Node-Y, receives the P_X, D_{G1}, and R_{G1} of T_{G1} from Node-X. T_G is validated at Node-Y. Then, Node-Y sends a lock-release request to Node-X.

Session_update:

Consider the user session P. The usage of S_P is the same as in the fully replicated protocol. A new array, SR_P, is used to indicate the nodes to which these preceding conflicting transactions stored in S_P are sent.

Let the update transaction, T_K, be executed in the user session P. The procedure stores in the session variable the USNs of transactions conflicting with T_K and nodes to which these transactions have been sent to.

```
for var = 1 to n, where n is the number of nodes
 for varp = 1 to n, where n is the number of nodes
  SP[var][varp] := Max(DK[var][varp], SP[var][varp])
  SRP[var][varp] := Max(RK[var][varp], SRP[var][varp])
```

A ROT may have read from data items written by more than one transaction. The TM lists the transactions with which the ROT conflicts. Let D-array and R-array corresponding to these conflicting transactions be D_1, D_2, D_3,..., D_M and R_1, R_2, R_3,..., R_M, respectively. The procedure stores in the session variable the USNs of transactions conflicting with T_K and nodes to which these transactions have been sent.

```
for var = 1 to n, where n is the number of nodes
 for varp = 1 to n, where n is the number of nodes
      SP[var][varp] := Max(D1[var][varp], D2[var][varp], D3[var][varp],...., 
DM[var][varp], SP[var][varp])
      SRP[var][varp] := Max(R1[var][varp], R2[var][varp], R3[var][varp],...., 
RM[var][varp], SRP[var][varp])
```

The *session_update* procedure for both global ROTs and local ROTs is the same, except that for global ROTs, the same procedure is executed both at the coordinator and participant nodes.

Delete_change-record:

The procedure is the same as in the case of the fully replicated system. But $M_X[y]$ indicates the USN of the latest transaction which was sent by Node-Y to other nodes and has been executed at those nodes as known by Node-X. The procedure given below stores in M-array, the USNs of transactions that have been executed at all nodes.

for var = 1 to n, where n is the number of nodes
 $M_X[var] := N_X[var][var]$
 for varp = 1 to n, where n is the number of nodes
  $M_X[var] := Min(N_X[varp][var], M_X[var])$
  where Min function returns the minimum integer of a set of integers.

This procedure is the same as in the fully replicated system. We have given here for the sake of completeness.

5.2 Causal multicast transmission of messages

The algorithm for the causal multicast differs from the causal broadcast, as causal multicast uses the R-array for the delivery of messages, instead of the D-array. The causal multicast mechanism for the delivery of message of T_K can be explained as follows:

Let us assume that a message of T_K is sent from Node-Y to Node-X. The procedure given below waits until all the messages of conflicting transactions of T_K that were sent to Node-X have been delivered at that node. After delivery of message, the state array at Node-X is updated to indicate the delivery event.

Wait until $N_X[x][1,2,3,\ldots,y-1,y+1,\ldots,n] \geq R_K[x][1,2,3,\ldots,y-1,y+1,\ldots,n]$
  then deliver the message of T_K at Node-X
After the delivery of message, $N_X[1,2,3,...,n][1,2,3,...,n]$ is updated as follows:
$N_X[x][y] := Max(N_X[x][y], R_K[x][y])$
$N_X[y][y] := Max(N_X[y][y], R_K[y][y])$
for var = 1 to n; where n is the number of nodes
if P_Y exists in the change-record
  then $N_X[y][var] := Max(N_X[y][var], P_Y[var])$

In the above code, if the present state of Node-X (denoted by $N_X[1,2,3,\ldots,n][1,2,3,\ldots,n]$) contains all the preceding conflicting transaction of T_K to be delivered at that node, then the change-record of T_K is delivered. To indicate the delivery event, the state array of Node-X is updated with the R-array of T_K. Later, Node-X's state array is updated with the P-array of T_K, if the P-array exists in the message.

5.3 Correctness Proof

First, we show that all the dependent messages are delivered causally ensuring the liveness property. Then, we show that the primary transaction protocol ensures 1SR and the session guarantee mechanism does not block any ROT indefinitely. Later, we show that the global ROT obtains a consistent state of the system.

Theorem 1: Algorithm ensures that all the messages are eventually delivered and the causal delivery is ensured using multicast primitives.

Proof: We prove the theorem with the following two claims:

Claim 1: Message delivery mechanism ensures the liveness property

On the contrary, let us assume that there exists a set of messages which are not delivered at Node-X. First, let us consider the non-delivered message Υ of the transaction, T_K, at Node-X which was sent from Node-Y. Υ may be a change-record, lock-grant, or acknowledgment message of T_K. If Υ is a lock-grant/acknowledgement message, then the reliable broadcast mechanism ensures that all messages that are sent from one node to another are delivered in the same order. On the other hand, if Υ is a change-record message, it can be of a local or global transaction. If Υ is a change-record message of a local transaction and its dependent change-record messages are also of the local transaction, then they will be eventually delivered due to the reliable broadcast. But, if T_K is a change-record of a global transaction, then it conflicts on a data item in custody of another node. Let us assume that T_K conflicts with T_L whose primary transaction was executed at Node-Z. Also, let T_L be a local transaction. The change-records of T_K and T_L are sent from Node-Y and Node-Z, respectively. At Node-X, the change-record of T_K is waiting for the change-record of T_L to be delivered at that node. If the change-record of T_L is actually sent to Node-X, then due to the reliable broadcast mechanism T_L is delivered at that node. But if T_L is not sent to Node-X, the R-array of T_K does not contain the USN of T_L corresponding to Node-

X. Therefore, T_K does not wait for T_L and T_K will be delivered eventually. The same holds true even if either T_K or T_L or both are global transactions. But this is contradictory to our earlier assumption.

Hence liveness is ensured.

Claim 2: Causal delivery of messages is ensured.

Let Node-Y and Node-Z be coordinators of T_K and T_L, respectively. Let us assume that both transactions have sent their change-records to Node-X. Let T_L precedes T_K ($T_L \rightarrow T_K$). Then, $R_K[x][z]$ will have the USN of T_L, or the later transaction. When the change-record of T_K is received at Node-X from Node-Y, it is delivered on the following condition has been satisfied.

"If $N_X[x][1,2,3,\ldots,y\text{-}1,y+1,\ldots,n] \geq R_K[x][1,2,3,\ldots,y\text{-}1,y+1,\ldots,n]$"

This means that $N_X[x][z]$ is at least the USN of T_L. Therefore, T_L has been delivered before T_K at Node-X. Therefore, causal delivery is ensured.

The proof is complete.

Lemma 1: The session guarantee mechanism does not block any ROT indefinitely.

Proof: Let us assume that the ROT, T_K, is executed at Node-X. Also let us assume that T_K is reading a data item in custody of Node-Y. The *session_read_check* procedure for T_K must wait until all the transactions in S-array, $S_P[y][1,2,3,\ldots,n]$, have been delivered at that node. On the contrary, let us assume that the procedure of

T_K is blocked because it is waiting for the transaction, T_L, which was not sent to Node-X from Node-Y. That is, $S_P[y][y] > N_X[x][y]$.

We know that the R-array associated with T_L, does indicate that T_L is not sent from Node-Y to Node-X. i.e., $R_L[x][y]$ does not contain the USN of T_L (but of the preceding conflicting transaction). Satisfying the following condition ensures that T_K does not wait for T_L.

"wait until ($N_X[x][1,2,3,\ldots,n] \geq SR_P[1,2,3,\ldots,n][1,2,3,\ldots,n]$)"

This is contradictory to our assumption that the execution of the ROT, T_K, is blocked due to T_L.

The proof is complete.

Theorem 2: A global ROT obtains a consistent view of the state of the system.

Proof: If two sub-transactions of a global transaction are executed at different nodes, then they must obtain a consistent state of the system. Let T_{K1} and T_{K2} be sub-transactions of the global ROT, T_K, executed at Node-X and Node-Y, respectively. Let Node-Y be the coordinator. Let us assume that the result obtained is not from a consistent snapshot of the system. This means that the state of Node-X is not consistent with the state of Node-Y, either with respect to sub-transaction T_{K1}, or sub-transaction T_{K2}.

We know that when the state of Node-X is compared with the state of Node-Y, locks are obtained on all the data items of the global transaction at both the nodes. For transaction T_{K1}, the positive result for the condition below ensures that all the dependent transactions of T_{K1} have been executed at Node-Y already.

```
if (D_K1[1,2,3,...,n] ≥ N_Y[y][1,2,3,...,n])
    for z =1 to n, where n is the number of nodes
    if (R_K1[y][z] ≥ N_Y[y][z] )
       then result is negative & exit
    end of for
else
    result is positive
```

While creating the D-array for a sub-transaction at a node, the conflict on a data item in custody of the local node is not found. This is because, locks on all the data items of the global transactions are released analogous to 2PL protocol. That is, a sub-transaction releases a lock only after the global transaction acquires all the locks via all its sub-transactions. Therefore, with respect to the execution of T_{K1}, the present state of Node-X and Node-Y are consistent. Similarly, satisfying the below condition ensures that all the preceding conflicting transactions of T_{K2} have been already executed at Node-X.

```
if (D_K2[1,2,3,...,n] ≥ N_X[x][1,2,3,...,n])
    for z =1 to n, where n is number of nodes
    if (R_K2[x][z] ≥ N_X[x][z] )
       then result is negative & exit
    end of for
else
    result is positive
```

So the state of Node-X and the state of Node-Y are consistent with respect to sub-transaction, T_{K2}. Upon satisfying the above two conditions, we can ensure that the global transaction, $T_{K,}$ obtains a consistent state of the system. But this is contradictory to our earlier assumption.

The proof is complete.

5.4 Discussion

The main advantage of the partial replicated system is that it requires communication only between a few peer nodes leading to timely relevant information to all the nodes. By the mechanism of the coordinator sending the relevant change-records to other nodes and the other nodes waiting for only those relevant change-records, the unnecessary information overload and waiting is reduced compared to the fully replicated system.

Chapter 6

Deadlocks

Deadlocks in the transaction processing context are situations in which two or more transactions are waiting indefinitely for one of the others to finish but none of them finishes. Any non-conservative locking mechanism is prone to the occurrence of deadlocks. The simplest illustration of a deadlock consists of two transactions, each holding a write lock on different data item and requesting a lock on a data item which is write locked by the other transaction. A deadlock is said to be distributed, if transactions executed at two or more nodes are involved in the deadlock.

6.1 Deadlocks in our protocol

Deadlocks are possible in our protocol, as it is basically a non-conservative 2PL. We first consider deadlocks in the fully replicated protocol and later in the partially replicated protocol. In the fully replicated protocol, local transactions are not involved in the deadlocks, as all the locks are requested in stage (1.a), atomically. This is shown in the following lemma.

Lemma 1: The local transactions are not involved in deadlocks.

Proof: Consider a local transaction, T_K. T_K requests locks only in stage (1.a). All the locks are obtained atomically and after this stage no locks are requested. Therefore, T_K does not wait for any other transactions. The proof is complete.

A global transaction executing at the coordinator node requests locks from other nodes. All the locks for a global transaction cannot be obtained in a single atomic step. Therefore, distributed deadlocks are possible. This can be illustrated with the following example.

Example 6.1: Consider the setup as shown in Figure 6.1. Node-X and Node-Y are custodians of data items b and c, respectively. Let T_K and T_L be the primary transactions with coordinators Node-X and Node-Y, respectively. T_K obtains a PT-X lock on data item b at Node-X in stage (1.a) of the primary transaction protocol. Similarly, T_L obtains a PT-X lock on data item c at Node-Y in stage (1.a) of its primary transaction protocol. T_K sends a PT-S lock-request to Node-Y for data item c. Similarly, T_L sends a PT-S lock-request to Node-X for data item b. T_K is waiting for T_L for a lock on data item c and T_L is waiting for T_K for a lock on data item b. This cyclic wait indicates the distributed deadlock.

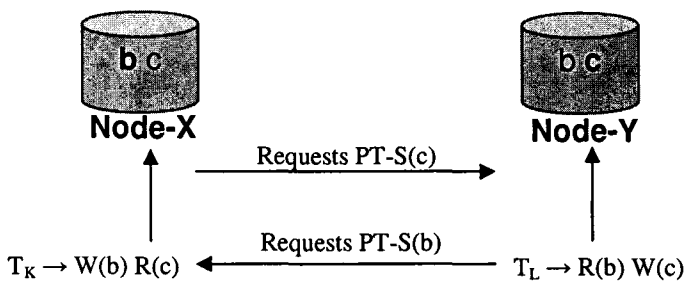

Figure 6.1 Illustrates distributed deadlocks due to global transactions

6.2 Distributed deadlocks

We design an algorithm which detects and resolves the deadlocks which is discussed in the following section.

6.2.1 Algorithm to detect and resolve deadlocks using wait for graphs

We first construct a Wait-For-Graph (WFG). It is a directed graph, where a T-node (transaction node) T_K represents a transaction, T_K, and its outgoing edges represent the lock-requests T_K has made to other nodes, and incoming edges represent lock-requests that other nodes have made for T_K. Depending upon the request on a data item of T_K, its T-node may have the following kind of edges:

- Unconnected edge: If T_K has requested a lock on a data item and is waiting for a response, then an unconnected outgoing edge (unconnected to any node at the other end) is drawn from the T-node of T_K.
- No outgoing edge: If T_K has already acquired a requested lock, then with respect to that request there is no outgoing edge in T-node of T_K.
- Connected edge: If T_K is waiting for a lock which is currently held by T_L, then a directed edge is drawn from the T-node of T_K to the T-node of T_L ($T_K \rightarrow T_L$). On the other hand, if T_L is waiting for a lock currently held by T_K, then a directed edge is drawn from the T-node of T_L to the T-node of T_K ($T_L \rightarrow T_K$).

A WFG consists of only global transactions in the system. That is, there are no local transactions in the WFG. At a node, there may be a set of WFGs corresponding to

different sets of transactions in the system. A WFG which contains the T-node of T_K is called the WFG of T_K (represented by WFG(T_K)). As the execution of the primary transactions are distributed (i.e., different primary transactions may be executed at different nodes), there is no total order in propagation of WFGs of these transactions. Therefore, at a certain snapshot of the system, WFG(T_K) present at different nodes may be different. Our algorithm is designed in such a way that, if a deadlock really exists in the system, the nodes in the system which constructs WFG(T_K) eventually will construct the same WFG for T_K. Therefore, a victim transaction selection and its abortion can be made deterministically.

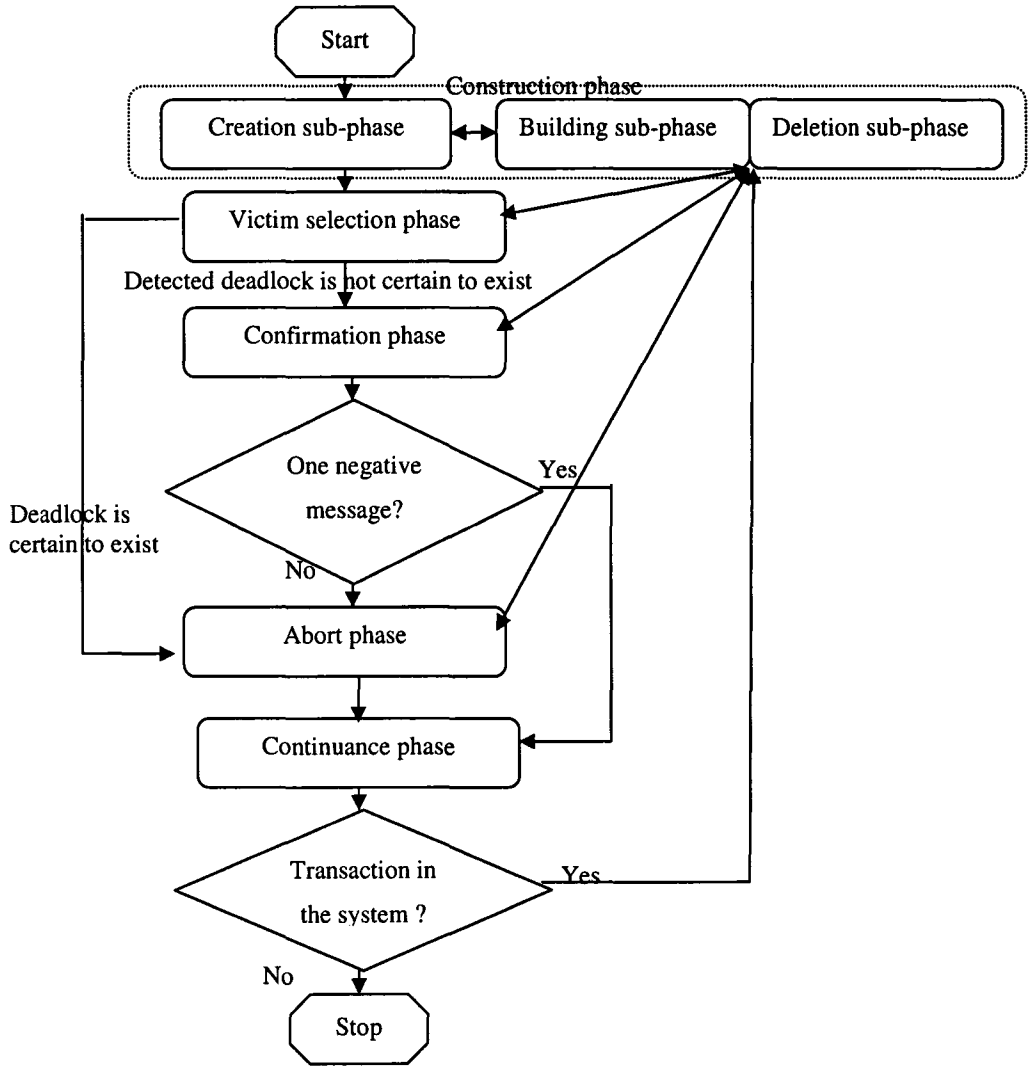

Figure 6.2 A Flow chart for detection and resolution of deadlocks

The algorithm for deadlock detection and resolution has five main phases (refer to the flow chart in Figure 6.2). The phases can be explained as follows:

1. Construction phase: Any change to a WFG is made in this phase. This phase is sub-divided into three sub-phases. When a global transaction is submitted to the system, it enters the first sub-phase. An algorithm may execute the second or the third sub-phases concurrent to the execution of any of the other phases. The sub-phases are classified as follows:
    a. Creation sub-phase: In this sub-phase, a new WFG of a transaction is created.
    b. Building sub-phase: In this sub-phase, a new T-node is added to the existing WFG by merging it with another WFG.
    c. Deletion sub-phase: In this sub-phase, an edge of a T-node or a T-node is deleted.
2. Victim selection phase: In this phase, one of the transactions in the deadlock cycle is selected as a victim transaction. If it is certain that a deadlock exists, then the algorithm goes to the abort phase directly.
3. Confirmation phase: The coordinator of the victim transaction starts the confirmation phase to check if the detected deadlock really exists. If it exists, the algorithm goes to the abort phase. Otherwise, it goes to the continuance phase.
4. Abort phase: The victim transaction is aborted.
5. Continuance phase: In this phase, the deadlock would have been successfully resolved.

Definitions:

We use the graphical interpretation of WFGs to explain the algorithm. The developer of an application is free to use his own data structure for implementation. Each node in the registry is assigned a priority which is unique in the entire registry. The priority is stored at every node in the registry. The T-node of T_K in WFG(T_K) at a node in the registry consists of the following variables:

1 Identifier of T-node: The T-node of T_K is identified by <Node-ID, Transaction-ID>. Node-ID is the identifier of the coordinator node of T_K. Transaction-ID is the identifier of T_K.

2 Outstanding requests: This request indicates either pessimistic or optimistic lock requests of T_K at participant nodes. If T_K has sent a request and is waiting for a response, then it is represented by an unconnected outgoing edge in WFG(T_K). On the other hand, if T_K knows that it is waiting for another transaction T_L, then the T-node of T_K is connected to the T-node of T_L with an edge represented by $T_K \rightarrow T_L$.

3 Locks acquired: No outgoing edge is associated with the T-node of T_K in WFG(T_K) for a lock already acquired by T_K. For each lock on data items, the type of lock is indicated (i.e., read lock or write lock).

4 Incoming requests: This request indicates either pessimistic or optimistic lock requests other transactions have made to data items on which T_K has already

acquired the locks. If T_L has requested a lock on a data item which is presently held by T_K, then a directed edge, $T_L \rightarrow T_K$, is added.

5 Total weight: Each T-node in the WFG is associated with a weight. The weight of T_K is represented by 'T_K.Weight'.

The weight assignment to a T-node in a WFG can be explained as follows:

A weight is assigned to a transaction in order to select a victim in a deadlock cycle. Weight is assigned to a transaction based on the type of its lock requests. A transaction requests read and write locks from other nodes. Each of these may be requested either pessimistically or optimistically. In stage (1.b) of the primary transaction protocol, the coordinator node knows all transaction requests to other nodes. Therefore, when a T-node of a WFG is created, the protocol knows all the transaction requests. For transaction T_K, weights are assigned based on type of request as follows:

1 Optimistic write: If transaction T_K has an outstanding optimistic write request on a data item, its T_K.Weight is incremented by 1. That is,

$T_K.Weight := T_K.Weight + 1$

2 Pessimistic write: If transaction T_K has an outstanding pessimistic write request on a data item, its T_K.Weight is incremented by 2. That is,

$T_K.Weight := T_K.Weight + 2$

3 Pessimistic read: If transaction T_K has an outstanding pessimistic read request on a data item, its T_K.Weight is incremented by 3. That is,

T_K.Weight := T_K.Weight + 3

4 Optimistic read: If transaction T_K has an outstanding optimistic read request on a data item, its T_K.Weight is incremented by 4. That is,

T_K.Weight := T_K.Weight + 4

Algorithm:

The algorithm for detection and resolution of deadlocks can be explained as follows:

1 Construction phase: This phase handles all the modifications of WFGs in the system. This is explained in the following sub-phases:

a) Creation sub-phase: Once a global transaction, T_K, is submitted to the coordinator, Node-X, the algorithm enters the creation sub-phase. This sub-phase can be explained from the view of the coordinator and participants of T_K (refer Figure 6.3).

View of the coordinator node of T_K, Node-X

i) Creating a T-node: When a global transaction, T_K, starts its execution in the system, coordinator Node-X creates the T-node of T_K. Corresponding to each outstanding request for the lock of T_K, an outgoing edge is associated with the T-node of T_K. This node creates the WFG(T_K).

ii) Initialization: When the T-node of T_K is created, it is assigned a weight. First, all the outstanding requests are listed. Then, a weight is assigned to the T-node. The assignment is done in stage (1.b) of the primary transaction protocol in the coordinator's view. Once a weight is assigned to a T-node of a transaction, it is not later altered.

iii) While sending the lock-request or the validation-request message of T_K: The coordinator, Node-X, while sending the lock-request or the validation request message of T_K, also sends the WFG(T_K) to the participants of T_K.

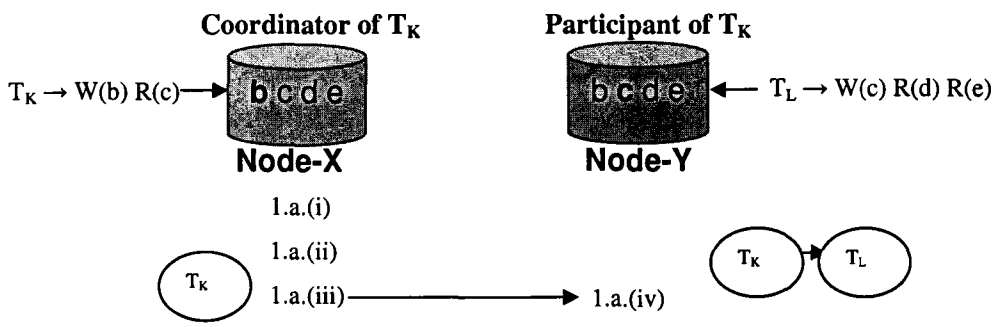

Figure 6.3 Illustrates the creation sub-phase of the algorithm

View of the participant node of T_K, Node-Y

iv) Upon receiving the WFG(T_K) with lock-request or validation request messages: If the requested lock is available at Node-Y, then the TM grants the lock. On the other hand, if T_K is waiting for another transaction to

release the lock, then either two WFGs are merged or an edge is added between T-nodes. This process can be explained as follows:

- Merging of two WFGs: If there are any common T-nodes between $WFG(T_K)$ and $WFG(T_L)$, then those WFGs are merged. Merging is performed by taking the union of T-nodes and edges of those two WFGs. If merging adds a T-node which has been previously deleted due to the delete-node message, then that T-node is not added into the WFG (similarly for the edge). We assume that delete-node and delete-edge messages sent or received at a node are archived until all the relevant nodes become aware of this event.

- Adding an edge between T-nodes: If T_K is waiting for a lock on a data item which is held by T_L and the present node is custodian of that data item, then the edge, $T_K \rightarrow T_L$, is added between their respective T-nodes. This is called a Wait-For-Relationship (WFR). Please note that even if a non-custodian node knows this WFR, it cannot add the edge. For example in Figure 6.3, only Node-Y, the custodian of data item c can add the edge, $T_K \rightarrow T_L$, as only that node knows if T_L is still holding the lock on data item c.

In $WFG(T_K)$, there is a set of transactions to which Node-Y is either a participant or a coordinator. The participant nodes for transactions to which Node-Y is coordinator of and the coordinators for transactions to

which Node-Y is participant of are called neighbors of Node-Y with respect to WFG(T_K). In Figure 6.4, the neighbors of Node-Y with respect to WFG(T_K) are Node-X and Node-Z.

After merging two WFGs of T_K or adding an edge in WFG(T_K), the new WFG(T_K) is sent to all the neighbors of Node-Y with respect to WFG(T_K).

b) Building sub-phase: In this sub-phase, the WFG created in the last sub-phase is modified by adding more T-nodes. This sub-phase can be explained from the view of the coordinator and participant nodes (refer Figure 6.4):

View of the participant node of T_K, Node-Y

i) Upon receiving WFG(T_K): When Node-Y receives the propagation of WFG(T_K) message from another node, it checks if there are any common T-nodes in any of the WFGs at that node and WFG(T_K). If there are any, then that WFG is merged with WFG(T_K). If the new WFG obtained by merging is different from both the WFGs, then the new WFG(T_K) is sent to the neighbors of Node-Y with respect to WFG(T_K). On the other hand, if there are no common T-nodes between WFGs, then WFG(T_K) is stored and no action is taken. In Figure 6.4, after WFG(T_K) is merger with another WFG, the new WFG(T_K) is sent to neighbors of Node-Y with respect to WFG(T_K) i.e., to Node-X and Node-Z.

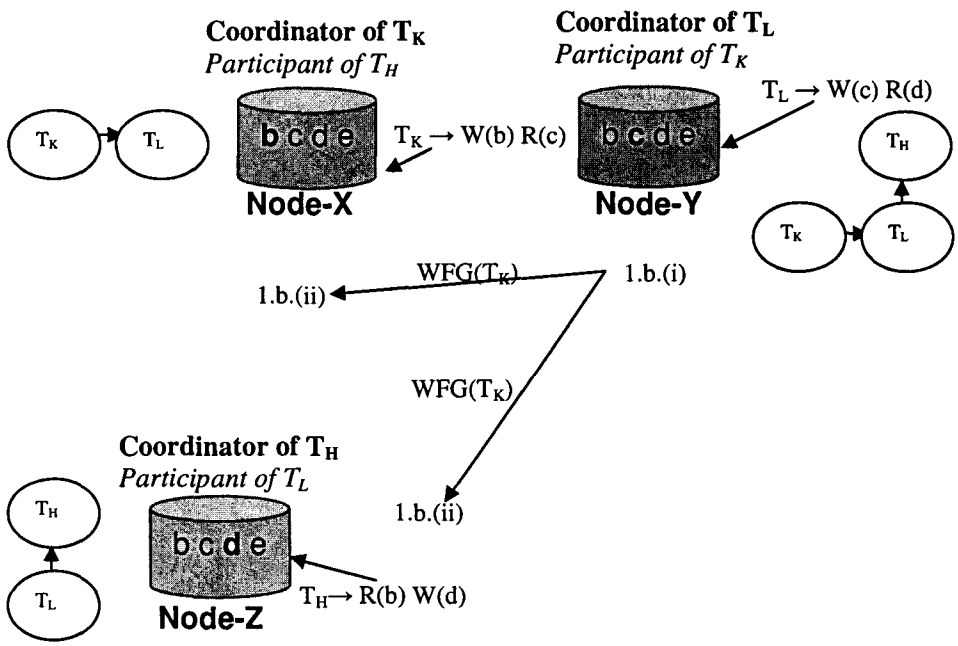

Figure 6.4 Illustrates the building sub-phase of the algorithm

View of the coordinator node of T_K, Node-X

ii) Upon receiving the propagation message containing WFG(T_K) or upon addition of a T-node of T_K to WFG(T_K): The procedure is similar to that of the participant.

c) Deletion sub-phase: Whenever a transaction finishes execution at the local node or a lock-grant message is received, its WFG is modified by deleting the T-node or edge, respectively. This can be explained from the view of the coordinator and participant of transaction T_L (refer Figure 6.5).

View of the coordinator node of T_L, Node-Y

i) Upon receiving the lock-grant message: When the coordinator of transaction T_L, Node-Y, obtains the lock-grant message of T_L which was held by T_H earlier, edge $T_L \rightarrow T_H$ in WFG(T_L) at that node is deleted. Later, a delete-edge $T_L \rightarrow T_H$ message to neighbors of Node-Y with respect to WFG(T_L) is sent.

ii) Upon commitment of a transaction: After T_L is successfully committed at Node-Y, delete-node of T_L message is sent to its neighbor of Node-Y with respect to WFG(T_L). Node-Y also deletes the T-node of T_L and all its outgoing edges from WFG(T_L).

We assume that delete-edge and delete-node messages are stored at all the nodes sending these messages until all other nodes in the registry become aware of these messages.

View of the participant node of T_L, Node-X

iii) On receiving delete-edge $T_L \rightarrow T_H$ or delete-node T_L messages: If Node-X has WFG containing either T_L or T_H, then on receiving these messages they are propagated to neighbors of Node-X with respect to WFG(T_L). Also, WFG(T_L) deletes edge $T_L \rightarrow T_H$ and T-node T_L corresponding to message types delete-edge $T_L \rightarrow T_H$ and delete-node T_L, respectively. If T-

node T_L is deleted, then all the outgoing edges connected to its T-node are deleted.

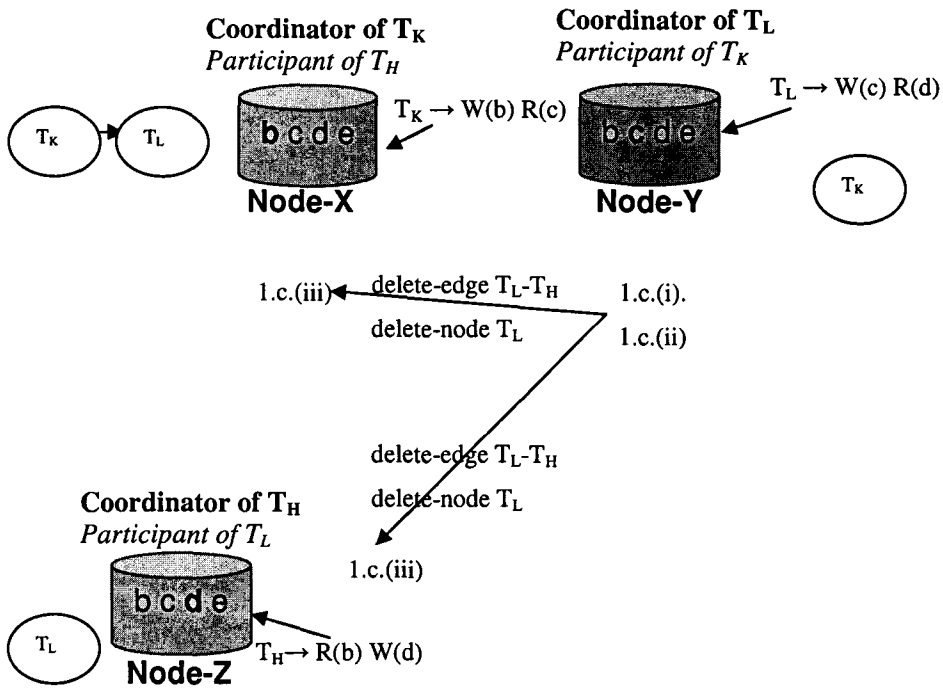

Figure 6.5 Illustrates the deletion sub-phase of the algorithm

2 Victim selection phase: Whenever a T-node or an edge is added to a WFG or two WFGs are merged, then the node at which WFG was modified checks for a deadlock cycle. If deadlock cycles are detected, then a deadlock cycle is selected for resolution as follows:

If more than two deadlocks are detected, then the cycle with smallest length is first selected for resolution. If the selected deadlock cycle is of length two, then

the algorithm goes to the abort phase directly where one of the transaction is aborted. Otherwise, in order to make sure that the deadlock exists, the algorithm goes to the confirmation phase.

The process of victim selection can be explained as follows:

Within a detected deadlock cycle, the transaction with the highest weight is selected as a victim. If two or more transactions have the same highest weight, then the transaction which starts its execution at the node in the registry with the lower priority is selected as a victim. If two or more transactions starts its execution at this lower priority coordinator, then the transaction with the highest transaction ID is selected as a victim.

Any node which detects a deadlock cycle selects a victim transaction. If the deadlock cycle is of length of two, the node sends the abort message to the coordinator node of the victim transaction. On the other hand, if the deadlock cycle is of length more than two, the node which detects deadlock cycle sends the initiate-confirmation message to the coordinator node of the victim transaction. The message contains the WFG of the deadlock cycle along with the victim T-node.

3 Confirmation phase: Upon receiving the initiate-confirmation message, the WFG is merged with the WFG at the local node. If there is a deadlock cycle, then the victim node (i.e., the coordinator of victim transaction) initiates the confirmation phase. This phase serves two purposes:

- Makes sure that the deadlock really exists. This is demonstrated with the following example.

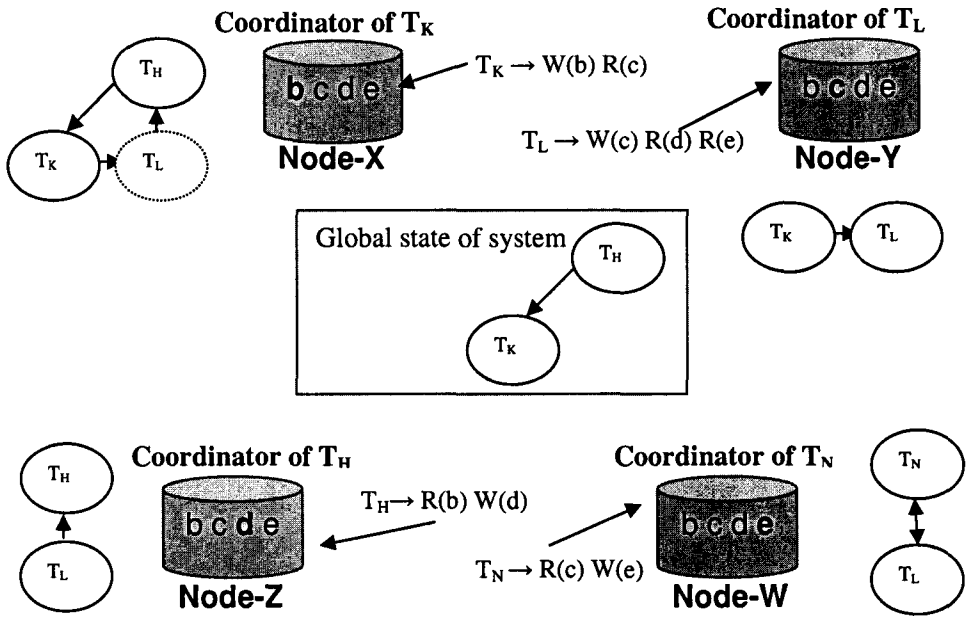

Figure 6.6 Illustrates that the deadlock cycle, $T_K \rightarrow T_L \rightarrow T_H \rightarrow T_K$, does not exist in the system at the time of resolution

Example 6.2: Consider the set up shown in Figure 6.6. Node-X detects deadlock cycle, $T_K \rightarrow T_L \rightarrow T_H \rightarrow T_K$, in $WFG(T_K)$ at that node. Node-X selects T_H as a victim of the deadlock cycle. Meanwhile, Node-W detects another deadlock cycle, $T_N \rightarrow T_L \rightarrow T_N$, and sends the abort of T_L message to the coordinator of T_L, Node-Y. Node-Y on receiving the message, aborts T_L. Node-X unaware of these events sends the abort of T_H message to the

coordinator of T_H, Node-Z. Now, Node-Z unaware that T_L has already been aborted by Node-W, aborts T_H. When T_H was aborted, there was no deadlock cycle containing T_H in the system. As no single node has the global view of the system a false deadlock exists in the system. The protocol must ensure that the deadlock really exist in the system at the time of resolution.

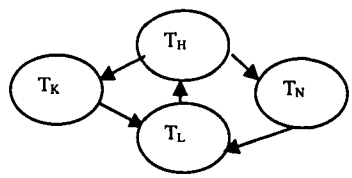

Figure 6.7 Illustrates that two deadlocks simultaneously exist in a WFG

- All the nodes in the registry abort the same transaction irrespective of the topology of the WFG of the victim T-node. This is demonstrated with the following example.

  Example 6.3: Consider the setup shown in Figure 6.7. It consists of two deadlock cycles, i.e., $T_K \rightarrow T_L \rightarrow T_H \rightarrow T_K$ and $T_N \rightarrow T_L \rightarrow T_H \rightarrow T_N$. As detection of deadlock is performed in a distributed fashion and there is no total ordering among the messages, no single node in the system may have the global view of the WFG as shown in the figure. One of the nodes in the registry which detects only the former deadlock cycle selects T_H as victim and aborts it. Later, other node which detects only the latter deadlock cycle, selects

T_L as a victim and aborts it. In summary, the algorithm aborts both T_H and T_L but aborting either one of them would have resolved the deadlock.

The confirmation phase can be explained from the view of the victim node and its d-participants (i.e., coordinators of other transactions in the deadlock cycle. Please refer Figure 6.8.)

View of the of victim node, coordinator of T_K, Node-X

a) Check if T_K is still running: First, the coordinator of the victim transaction, T_K, checks if T_K is still running. At any point in the execution of this phase, if T_K is aborted, then victim node Node-X sends the delete-node T_K message to d-participants, if not already sent, and goes to the continuance phase.

b) Sends the confirmation-request message: The victim, Node-X, sends the confirmation-request message to its d-participants along with T_K's information (i.e., weight of the transaction, Transaction ID, its coordinator's priority).

c) Upon receiving confirmation messages: If the victim node receives a positive-confirmation message from all its d-participants, then the algorithm goes to the abort phase where the victim transaction is aborted. On the other hand, if Node-Y receives the negative-confirmation message from any one of the d-participant, then it sends no-deadlock cycle to all the d-participants and the algorithm goes to the continuance phase.

d) Victim node receives the information about other victim nodes: This means that its d-participants are part of another deadlock cycle. Node-X sends its WFG to the victim of another deadlock cycle. Another victim node upon receiving the WFG from the victim node constructs a global WFG and resolves deadlocks one after another. Among the set of victim nodes, one final victim node is selected and the abort message is sent to the victim's coordinator node. Then, it goes to the abort phase.

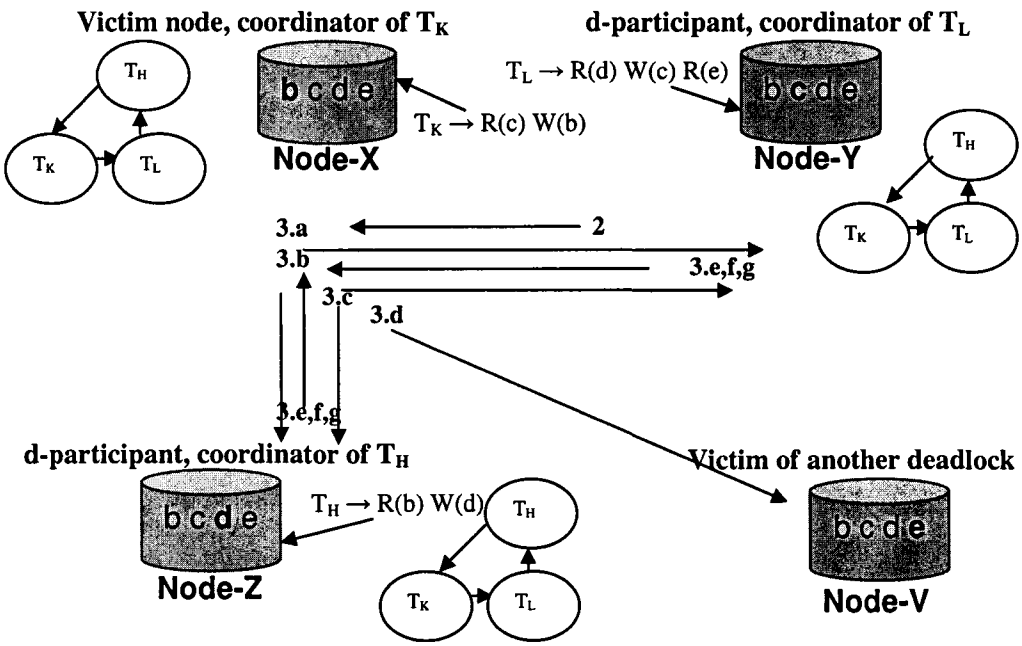

Figure 6.8 Illustrates the confirmation phase of the deadlock algorithm

View of the d-participant - T_L, Node-Y

On receiving the confirmation-request message, one of the following cases can occur:

e) T_L has not sent a positive-confirmation message to another deadlock cycle: The information about the victim transaction (i.e., weight of the transaction, Transaction ID, its coordinator's priority) is stored and a positive-confirmation message is sent to victim node.

f) T_L has sent a positive-confirmation message to the victim of another deadlock cycle: This means that T_L is already involved in another deadlock cycle. The coordinator of T_L sends the information about the victim node of another deadlock cycle to which it had sent the positive-confirmation message.

g) T_L has already started a confirmation phase: This means that T_L is a victim transaction of another deadlock cycle and is waiting for a confirmation message from at least one d-participant. The coordinator of T_L appends T_L's information (i.e., weight, ID, coordinator's priority) and forwards it to the d-participants to which it is waiting for. If the coordinator of T_L receives it own probe, using victim selection procedure, it selects one final victim among the set of victim transactions and sends the abort message to the final victim node.

On the other hand, if Node-Y had already sent a positive-acknowledgement message to a victim node earlier and now receives the no-deadlock message from

it, then the positive-confirmation message is acknowledged. This means that if T_L at Node-Y receives any new confirmation-request from another victim transaction, then the positive-confirmation message can be granted to the new request.

4. Abort phase: The coordinator of the victim transaction starts this phase on receiving the abort T_K message or as per the decision made by the victim node in the confirmation phase. It aborts the victim transaction, T_K, deletes the corresponding T-node from WFG(T_K) and propagates delete T-node T_K message to d-participants in the deadlock cycle. This can be explained from the view of the coordinator and the participant of T_K (refer Figure 6.9).

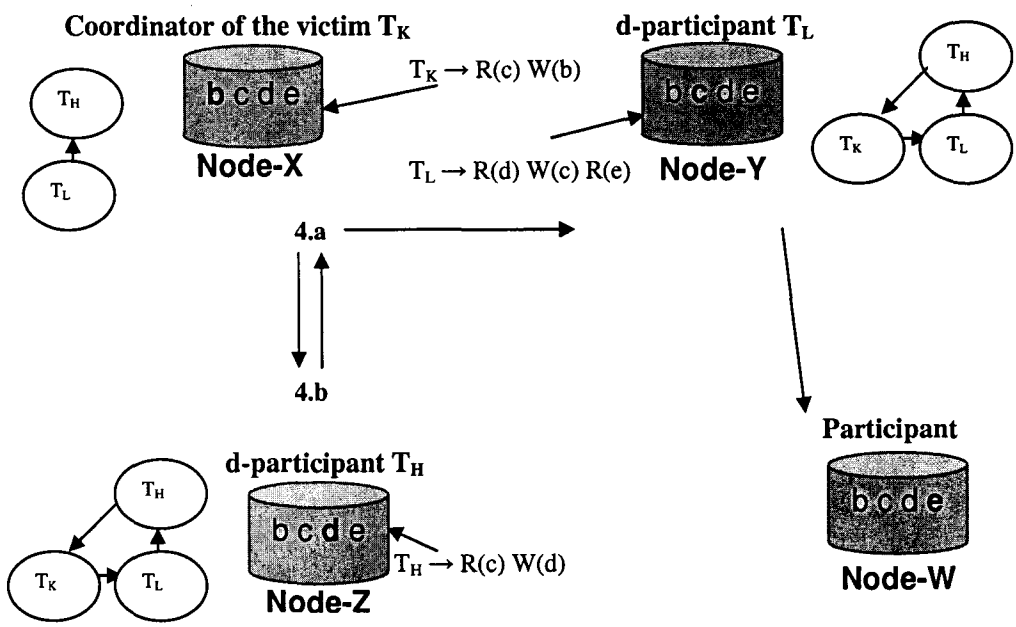

Figure 6.9 Illustrates the abort phase of the deadlock algorithm

View of the coordinator of T_K, Node-X

a) Node-X deletes the T-node of T_K and all its outgoing edges in WFG(T_K). It sends the delete-node of T_K message to all its d-participants of deadlock cycle.

View of the d-participant, the coordinator of T_L, Node-Y

b) Upon receiving the delete-node T_K messages for transaction T_K, Node-Y deletes the T-node of T_K and all its connecting edges. Then, Node-Y similarly propagates the delete-node T_K message to its participant nodes.

5 Continuance phase: During this phase, at least one node in the deadlock cycle of T_L knows that the deadlock does not exist. Later, all the d-participants become aware of this through the message propagation. Hence the deadlock is successfully resolved.

6 As all the transactions in a WFG finishes execution, eventually that WFG at all the nodes are deleted. Once all the transactions in the system finish execution, the algorithm comes to a halt.

6.3 Correctness proof

First, we show that if a deadlock cycle really exists in the system, then it will be detected eventually. Then, we show that a deadlock victim selection is made deterministically. Lastly, we show that if a transaction is aborted to resolve a deadlock, then such a deadlock really exists at the time of resolution in the system.

Theorem 1: At least one of the nodes in the registry which is involved in a deadlock cycle detects it.

Proof: Let us assume that a deadlock really exists in the system. Then, there exists a cyclic wait among transactions which executes at two or more different nodes. We know that at a node, if some global transaction is waiting for a lock which is held by another global transaction, then a WFG containing those two transactions is constructed at that node. Later, any modification to this WFG is propagated to the node's neighboring nodes with respect to that WFG. By this method of propagation, nodes which are coordinators of the global transactions in a deadlock will eventually receive the propagation message. Hence, any one of these nodes can detect the deadlock cycle.

Proof is complete.

Lemma 1: For a given deadlock cycle, if the algorithm detects the same deadlock at more than one d-participant node, then all of them will select the same victim.

Proof: Let us assume that a deadlock really exists. Then by Theorem-1, at least one of the nodes in the registry detects the deadlock. As message propagation and merging of messages is uniform, all the nodes which construct the WFG for a transaction will construct the same WFG for that transaction, eventually. For the selection of the victim transaction, the factors considered are weight of the victim

transaction, priority of the node in the registry and transaction ID. No two transactions in the system can have all these parameters in common. Hence the victim transaction is selected deterministically.

Theorem 2: Only true deadlocks are resolved. That is, if a victim transaction is aborted in a deadlock cycle, then such a deadlock really exists in the system at the time of resolution.

Proof: Let us assume that a victim transaction is selected by one of the nodes in the deadlock cycle. During the confirmation phase, the algorithm checks if all the transactions which are a part of the deadlock cycle are still running, and are not involved in the resolution of another deadlock cycle simultaneously. The participant of the deadlock cycle sends a positive-confirmation message to the victim node, only if the transaction is still running and is not involved in a deadlock with any other transaction in the system. Therefore, one victim in a cycle is uniquely selected and aborted. In this way deadlock resolution is deterministic.

6.4 Discussion

A weight is assigned to a T-node of a WFG based on the ranking for request types of the transaction. The ranking for each type of requests can be explained as follows:

1. Optimistic write request: The transaction with this operation has already finished the execution phase of the primary transaction. The basic protocol does not abort this transaction. Therefore, this kind of request is assigned the lowest rank.

2. Pessimistic write request: The transaction with this operation can both read and write a data item requested from the other node. Therefore, this kind of request is assigned the lower rank than transaction with pessimistic read.

3. Pessimistic read request: As the transaction with this operation only reads a data item, this request is assigned the higher rank than pessimistic write request.

4. Optimistic read request: The transaction with this operation has already finished the execution phase of the primary transaction. Even if another transaction which is conflicting with this transaction in the global deadlock cycle is aborted, this transaction may still be aborted by the replication protocol due to conflicts. That is, a preceding conflicting transaction, which has already committed, may have written a data item read by the transaction. This request is assigned the highest rank.

The algorithm is designed in such a way that the deadlock detection is performed by the propagation of messages. That is, all nodes in the system do not receive these messages in the same order. Also, these propagation messages are sent only among neighboring nodes. Once a deadlock is detected by any one of the nodes, the victim is selected and the algorithm resolves the deadlock in a coordinated fashion. The main benefit of our algorithm is that it does not abort any transaction due to a false

deadlock (i.e., the deadlocks which do not exist at the time of resolution). Our algorithm minimizes the number of transactions aborted to resolve the deadlocks. If the victims of two or more deadlocks are selected such that aborting one of the victims resolves another deadlock also, then only one transaction is aborted. The algorithm resolves the complex configuration of deadlocks deterministically.

The occurrence of deadlocks in our algorithm is directly proportional to the number of global transactions in the system. It also depends on the number of factors, such as the data item locking time, the number of data items locked by the transactions and if the lock on the data items has been requested pessimistically or optimistically. The global transactions having only the S-optimistic-request do not cause deadlocks in the system. In general, if all the remote locks of the global transactions are requested optimistically, it reduces the chances of deadlocks.

If most of the deadlocks in the system are of length two as in conventional database system [GJ81], then, our algorithm performs efficiently. But as the length of the deadlock cycle increases, the number of messages propagated in the system increases and algorithm's performance decreases.

The main differences between the protocols for the fully replicated system and the partially replicated system are the decision of when to request the locks on data items and the nodes to which the change-records of these transactions are sent. The decision of when to request the locks can affect the likeliness of the occurrence of deadlocks but does not affect the protocol for deadlock detection and resolution. Another

difference is that the global ROTs may be involved in the distributed deadlocks in partially replicated system. This is because a global update transaction may be holding a lock of a data item and requesting a lock from another node which is held by a sub-transaction of a global ROT. Another sub-transaction of the same global transaction, in turn, may be waiting for the lock held by the global update transaction, which creates a cyclic wait of locks among the transactions. This problem can be solved by aborting the global ROTs by timeout mechanism. That is, an abort message is sent to the coordinator of the global ROT and the coordinator aborts the transaction.

Chapter 7

Web Service discovery using UDDI

In this chapter, we discuss the discovery of Web Services using UDDI.

7.1 Introduction

UDDI registries are accessed by service providers and service requestors. In the registry, data items are only owned by service providers. The service requesters do not own any data item. They only query for services. A service provider may also query services published by another service provider. Ideally, a service provider in a session looking for services updated by other service providers expects to see those updates in the order of their execution. The protocol discussed in chapter 4 does not ensure this. This can be illustrated with the following example.

Example 7.1: Consider service providers, namely, P_1 and P_2, in the UDDI registry which own business entities BE_1 and BE_2, respectively (refer to Figure 7.1 for the data structure of the business entities). These providers are in a close collaboration on certain project. The service providers P_1 and P_2 execute T_1, T_3, T_4 and T_2, T_5, T_6, T_7, T_8, respectively, in their sessions. Similarly, service requestor R_1 executes T_9 and T_{10} in a session (refer Table 2). The sessions of P_1, P_2, and R_1 are S_1, S_2, and S_3, respectively. We assume that in a given session, transactions are ordered serially. A transaction in a session starts its execution only after the previous transaction

submitted to the system in that session commits. That is, in Table 2, T_3 and T_4 belonging to session S_1 are executed such that, T_4 starts its execution only after T_3 commits. The notation 'T_5:= Read(BS_{11}):T_4' means that T_5 reads data item BS_{11} written by T_4.

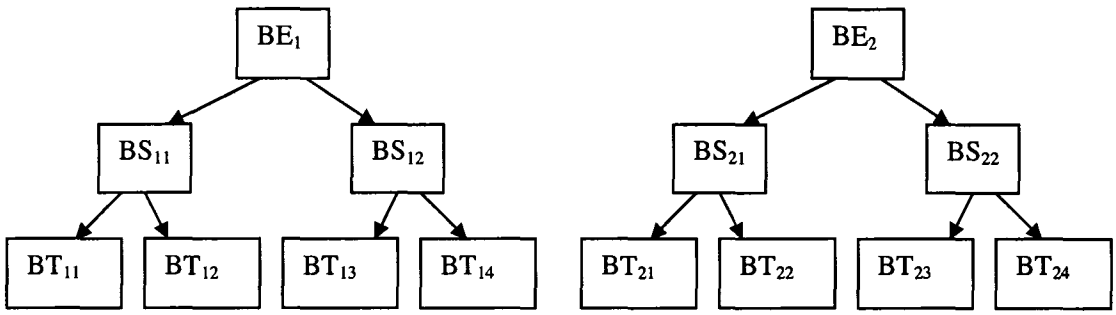

Figure 7.1 Illustrates the data structure of BE_1 and BE_2

The internal organization of entities in the registry is such that entities of the same service provider are in custody of different nodes. That is, entities BE_1, BS_{11}, BS_{12}, BT_{11}, BT_{12}, BT_{13}, BT_{14}, tM_1 owned by P_1 may be in the custody of different nodes in the registry. Similarly, entities owned by P_2 may be in custody of different nodes. Transactions of the same session may be executed at different nodes. (That is, their coordinators are different nodes.) Please note that T_1 and T_3 are conflicting transactions, as they update the same data item BS_{12}. Similarly, T_1 and T_4 are conflicting transactions. But, T_3 and T_4 are non-conflicting transactions.

S_1 of P_1 (owner of BE_1)	S_2 of P_2 (owner of BE_2)	S_3 of R_1 (service requestor)
T_1:= Insert(BE_1, BS_{11}, BS_{12}, BT_{11}, BT_{12}, tM_1)	T_2 := Insert(BE_2, BS_{21}, BS_{22}, BT_{21}, BT_{22},tM_2)	
T_3:= Update(BS_{12})		
T_4:= Update(BS_{11})		
	T_5:= Read(BS_{11}):T_4	
	T_6:= Read(BS_{12}):T_1	
	T_7:= Update(BS_{21})	
	T_8:= Read(BS_{12}):T_3	T_9:= Read(BS_{21}):T_7
		T_{10}:= Read(BS_{12}):T_1

Table 2 Illustrates the execution of the transactions by service providers, P_1 and P_2, and service requestor R_1

In session S_2, T_5, and T_6 read from T_4, and T_1, respectively. T_4 and T_1 were executed in session S_1. That is, the first read operation of S_2 reads from the latest transaction, T_4, and the later operation reads from the older transaction, T_1 (instead of T_3). This transaction inversion in S_2 happens because the replication protocol discussed in chapter 4 does not impose order on the execution of T_3 and T_4 at a node, as they are

non-conflicting transactions. That is, service provider P_2 in S_2 does not read an increasing state of session S_1. This could affect subsequent operations in S_2. For example, if T_6 had read BS_{12} updated by T_3, instead of T_1, P_2 might not have updated BS_{21} in T_7.

Note that in the example, there is a dependency between the operations of S_1 and S_2. Session S_2 reads a data item updated in session S_1 and later executes its operation. That is, T_5 of S_2 reads from T_4 of S_1. Therefore, there is an indirect dependency between T_7 and T_3, as T_3 is ordered before T_4, in S_1. In session S_3, the first operation reads from T_7, and the later operation reads from T_1 (instead of T_3). This is an inconsistent view, as order of execution of transactions at any node should be $T_1 \rightarrow T_3 \rightarrow T_7$. That is, if an operation in session reads from T_7, the next operation should at least read from T_3, in-order to provide an increasing view of the system. Our aim is to provide a mechanism to take care of these inconsistencies.

This problem can be handled by ensuring strong session 1SR [DS04]. A simple method to ensure strong session 1SR is to induce a conflict between every consecutive transaction of the same session. In this method each session contains a session variable called the ticket data item. Every transaction in a session reads and updates the same ticket data item. Therefore, when a transaction is executed using the replication protocol in chapter 4, every two consecutive transactions of the same session conflict with each other. Also, the message propagation mechanism in chapter 4 will ensure that all the update transactions in a session will be delivered and

executed at all nodes in the same order. When this method is employed in Example 7.1, the execution of T_4 at a node implies that T_3 has already committed at that node. In session S_2, as T_6 is executed after T_5, T_6 would read from T_3 instead of T_1. Also, T_6 which is a ROT when submitted to the system using the ticket method would induce a direct visible conflict between T_3 and T_7. Therefore, another session would read the transactions in order, T_3 followed by T_7.

The main disadvantages of this method are the following:

- Increases the number of global transactions in the system: Consider a local transaction submitted to the system. The ticket method enforces this transaction to read and update the ticket data item. As the ticket data item may be in custody of a node, other than the coordinator node, such a local transaction is converted to a global transaction.
- Increases the number of update transactions in the system: As all the transactions in a session update the ticket data item, this method converts all ROTs to update transactions.

The above two factors decrease the performance of the system. We propose a better solution, which is presented in the following section.

7.2 Protocol to ensure strong session 1SR

In this chapter, we extend the mechanism discussed in chapter 4 to ensure strong session 1SR. The aim of the protocol is to execute all the operations (i.e.,

transactions) of the session in the same order at all the nodes. The new protocol can be explained as follows:

The protocol requires the additional variables to ensure strong session 1SR. The two variables required are:

- Variables in the input stream of the user session

  An operation in a session is uniquely identified by <Session-ID, Operation-ID>. This indicates the last operation performed by the user in the session. This identifier is included along with S-array (introduced in chapter 4) in the input stream of the user session.

- Variables at a node in the registry

  At every node, along with the state array (N-array), a session array (SN-array) is also stored. The session array (denoted by SN[1,2,3,...,q]) indicates the latest operations in sessions 1,2,3,...,q that have been executed at that node. That is, one dimensional array, SN_X[1,2,3,...,q], denotes operations of sessions 1,2,3,...,q that have been executed at Node-X. Therefore, SN_X[p] indicates the last operation of the session p that has been executed at Node-X.

Execution of a transaction:

The protocol discussed in chapter 4, allows two consecutive non-conflicting update transactions in the same session to be executed at different nodes without any

coordination. That is, when a transaction of a session is executed at a node, its previous transaction in the same session may not be executed at that node. But, this violates the criterion of strong session 1SR. In this chapter, the protocol is extended to ensure strong session 1SR.

The session variables are modified before and after the execution of both the primary transaction of update transaction and the ROT. This procedure does not modify the refresh transaction of an update transaction in chapter 4. Note that these procedures are executed in addition to the procedures in chapter 4. That is, we assume that the underlying mechanism ensures guarantee provided in chapter 4.

Session_update:

This procedure is executed after the execution of the transaction in a session at a node.

Let us assume that a transaction in a session is executed at a node. Let the transaction be m^{th} operation (i.e., transaction) in a session, Sid. After the transaction is executed at Node-X, the session variable stored at that node and session variable in the user input stream are updated. That is,

SN_X[Sid] := m // session variable stored at Node-X
<Sid, m> // session variable stored in the user input stream

In the above code, SN_X[Sid] indicates the operation that was recently executed in the session, Sid, at Node-X. The user input stream is updated to indicate that m^{th} transaction of the session has been executed.

Session_read:

This procedure is executed before the execution of the ROT and the primary transaction of an update transaction in a session at a node. Please note that for the protocol in chapter 4, the *session_read* procedure is not performed before the execution of the primary transaction of an update transaction.

Let us assume that the session, Sid, has already executed a transaction which is m^{th} operation in that session. The session variable in the input stream of the user session contains the identifier of m^{th} transaction (<Sid, m>) executed in that session. The next transaction, $(m+1)^{th}$ operation, in the same session can execute at its coordinator node, if that node has already executed m^{th} operation of the session. Otherwise, $(m+1)^{th}$ operation waits until m^{th} operation of the session is committed at that node. That is,

Wait until ($SN_X[Sid] \geq m$)
then execute the $(m+1)^{th}$ transaction

In the above code, $SN_X[Sid]$ indicates the operation that was last executed in the session, Sid, at Node-X. It waits until m^{th} transaction of the session is executed at that node.

The creation of the change-record:

After the primary transaction commits at its coordinator node, the change-record of the transaction is created at the coordinator node. The creation of the change-record

for an update transaction is similar to that discussed in chapter 4. In an effort to ensure strong session 1SR, the identifier of the operation within the session, <Session-ID, Operation-ID>, is added to the change-record of the transaction.

The major difference in creation of the change-record compared to chapter 4 is that a change-record is created even for a ROT. The procedure to create a change-record for a ROT is similar to the update transaction explained above. The ROT is also assigned an identifier <Session-ID, Operation-ID>, which indicates the operation number in its session.

After the delivery of the change-record of a ROT at a node, it is neither executed nor stored at that node. The change-record of an update transaction is implemented with the refresh transaction protocol of chapter 4.

Propagation of the change-record:

The message propagation mechanism discussed in chapter 4 avoids the false causality. As a result, a change-record of a transaction to be delivered waits only for the preceding conflicting transactions in the system. We modify the session guarantee mechanism, such that a change-record of a transaction in a session not only waits for its preceding conflicting transactions, but also for previous transactions executed in the same session which are non-conflicting. This condition is required to ensure strong session 1SR, as for every pair of committed transactions in a session, the first transaction's commit should precede the second transaction's commit in one copy history. Please note that this process of waiting for the previous transaction is not

only for an update transaction but also for a ROT. This kind of guarantee is indeed required to ensure a consistent view between multiple sessions, as indicated in Example 7.1. In the example, the service requestor session, S_3, does not have the global view of all the operations of S_1 and S_2. The transaction execution discussed in chapter 4 is distributed, facilitating the primary transaction to execute at any node, which suits the configuration of UDDI. As a result, T_9 and T_{10} of S_3 read from T_7 and T_1, respectively, even though the primary transaction of T_3 has committed before the primary transaction of T_7 could start. In [DS04], they do not consider the issue of consistent view in multiple sessions, as they consider the centralized system where all the update transactions are executed at a single node. Commit order of all the update transactions are fixed in the global serialization graph at one node, and they are propagated and committed in the same order at all the nodes. As a result, ensuring guarantees between multiple sessions is simple in [DS04].

The new message propagation mechanism to ensure the causal delivery of messages, such that strong session 1SR criterion is satisfied can be explained as follows:

Let the transaction, T_K, be m^{th} operation in session, Sid. The change-record of T_K is sent from Node-X to Node-Y. The procedure given below waits until all the conflicting transaction of T_K and previous transactions in that session, Sid, have been delivered at Node-Y. After delivery, the variables at Node-Y are updated to indicate the delivery event.

Wait until ($N_Y[y][1,2,3,\ldots,x-1,x+1,\ldots,n] \geq D_{K\text{-}COL}[1,2,3,\ldots,x-1,x+1,\ldots,n]$
&& $SN_Y[Sid] = m-1$)
then deliver the change-record of T_K
$N_Y[y][x] := P_X[x]$
$SN_Y[Sid] := m$

In the above code, $D_{K\text{-}COL}[1,2,3,\ldots,n]$ contains the maximum USNs in each column of $D_K[1,2,3,\ldots,n][\ 1,2,3,\ldots,n]$. That is, the D-array of two dimensions is converted to one dimension, maintaining USNs of all the preceding conflicting transactions. The first condition ensures that the state array of Node-Y is compared with the D-array of T_K. This condition is similar to the condition in the message propagation mechanism in chapter 4. The second condition ensures that the last operation of the same session has been executed at the present node before delivery. This is a new condition which is required to ensure strong session 1SR. After the delivery of a message the state array of Node-Y is updated to indicate the event of message delivery. Then, the session array at the node is updated to indicate that a change-record of a transaction has been delivered.

7.3 Correctness proof

We have to show that our present extension to protocol in chapter 4 satisfies strong session 1SR.

Theorem 1: The algorithm ensures strong session 1SR.

Proof: Let T_K and T_L be executed in the same session. In one copy history T_K precedes T_L ($T_K \rightarrow T_L$), if one of the following two conditions is satisfied:

(1) T_K conflicts with T_L and precedes it.

(2) T_K commits before T_L starts its first operation of the transaction.

The protocol in chapter 4 ensures that all conflicting transactions are delivered in the same order at all the nodes. Later, all these transactions are executed in the same order. As the protocol in this chapter is built on the protocol in chapter 4, ordering imposed on the transactions is maintained. In message propagation mechanism, the first condition is the same as in chapter 4. Therefore, (1) is satisfied.

We have to show that if T_K commits before T_L starts its execution, then they are executed in the same order at all the nodes. Both T_K and T_L can either be update or read only transactions. Therefore, we have the following cases:

Case 1: Both T_K and T_L are update transactions.

Before execution of each primary transaction of T_L at the coordinator, the *session_read* procedure ensures that T_K has been executed at that node. Later, the message propagation mechanism ensures that the change-record of T_L is delivered only after the delivery of the change-record of T_K. Hence, these two transactions are delivered and executed in the same order at all the nodes.

Case 2: At least one of them is a ROT.

Let T_K be a ROT. After its execution, a change-record is created for the ROT, T_K, and broadcast to all nodes. Later, T_L is executed at its coordinator node only if the change-record of T_K has been implemented at that node. The message propagation mechanism ensures that these two transactions are delivered in the same order at all the nodes. Hence, when T_L is executed at a node all its previous transactions in that session have been executed at that node.

Case 3: Both T_K and T_L are ROTs.

We know that T_L executes at a node only after the change-record of T_K is delivered at that node. Also, T_K is delivered at that node only after all the dependent change-records in other sessions have been delivered at that node. Similarly, these dependent change-records are delivered only after its dependent change-records have been delivered. Therefore, when T_L is delivered at that node, all its indirectly dependent transactions have been delivered.

From above three cases we can conclude that all the transactions in the same session and conflicting transactions in multiple sessions are executed in the same order at all the nodes.

The proof is complete.

7.4 Discussion

The replication protocol designed in chapter 4 orders only the conflicting transactions at all nodes in the system. A user session in the system views only these conflicting

transactions in increasing order of their execution. Therefore, the non-conflicting transactions executed in the same session may not be seen in the same order by another session. This violates strong session 1SR as the transactions executed in the same session are not ordered by their commit time at all nodes.

In this chapter, we have extended the replication protocol in chapter 4 to ensure strong session 1SR. The execution of transactions in our system is distributed. That is, different transaction in a session can be executed at different nodes. This holds for both update and read only transactions. An interesting feature is that the operations executed in two different sessions may be ordered globally, due to read only transactions executed in a session. Therefore, any session reading these transactions should obtain a consistent global view. In order to provide a consistent global view, change-records are generated even for the read only transactions. The ordered delivery of these change-records at all nodes makes an indirect conflict visible to another session. The main disadvantage of this method is that, as change-records are created even for a read only transaction, the number of messages propagated in the system will be very large. Therefore, the system performance decreases.

The performance of the system can be increased for applications where one organization does not have visibility of operations of second organization with the third organization. That is, in a loosely coupled application, visibility is such that it supports an increasing view of peer to peer organizations. A suitable correctness criterion in such an application would be to provide an increasing view of one session

to another session. That is, a session provides a Monotonic Reads session guarantees to a session, if those operations were read from the same session. On the other hand, if those operations were read from different sessions, then it does not ensure Monotonic Reads session guarantees to the session considered above. This criterion is stronger than the session guarantee discussed in chapter 4, as it orders even the non-conflicting transactions which are executed in the same session, but weaker than strong session 1SR, as two ROTs are not ordered if the data items were read from transactions updated by different sessions. The implementation of this criterion is simple, as it is the same as in the above method, except that, there is no need to create change-records for read only transactions.

Chapter 8

Conclusion

The major contribution of this thesis is the replication protocol for both fully and partially replicated systems and the deadlock algorithm. The replication algorithm is fully distributed as transactions can be executed at any node. The algorithm is such that an update transaction can be executed with the minimum synchronization requirement, providing lower response time. The session guarantee mechanism allows a user to execute a transaction at any node and ensures that a user in a session obtains an increasing state of the registry. This increases availability of the system and provides an efficient load balancing. Even though our protocol requires high communication volume and storage capacity, we provide a fine grained session guarantee with a distributed execution. All these resource are required as we are using each of them to increase the performance of the system. The major contribution in the partially replicated protocol is to decrease the unnecessary communication cost by employing the multicast mechanism. Then the violation of the liveness property created due to multicast affects the basic replication protocol and the session guarantee mechanism. We provide a solution to above problems with suitable extensions.

The major contribution of our deadlock algorithm is to detect and resolve a deadlock without aborting a transaction, which is not a part of deadlock cycle at the time of

resolution. The algorithm is capable of handling the deadlock cycles with complex configuration.

Lastly, we extend the session guarantee mechanism to ensure strong session 1SR which is required in UDDI context. Our major contribution is providing strong session 1SR in distributed environment where a transaction in a session can be executed at any node.

Some of the future directions of our research are as follows:

- In our work we have consider replication of data items in a single registry. In UDDI registries, data items may be replicated in more than one registry. This kind of replication imposes new kind of restrictions, such as autonomy, freshness requirement etc. It is a reasonable requirement to give autonomy of different business registry to their respective organizations, as there is lack of trust among them. Achieving one copy serializability at a global level in a federated registry is an interesting problem. Also, if an organization replicates data items in its registry using publish-subscribe mechanism, it is difficult to ensure any known correctness criteria. This is because the underlying message propagation mechanism does not ensure synchronized delivery. Devising a suitable correctness criterion in such an environment is an interesting problem.
- Presently, UDDI registries are only used for answering simple queries. As UDDI matures, it will be used for the execution of complex queries in Web Service

composition. Usually, execution of complex queries decreases the system performance, as data items have to be locked for a longer duration of time. Composing partial results from different queries would be a better alternative. A scheme to provide a consistent snapshot of the registry by composing the partial results of different queries is an interesting problem.

Bibliography

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno and Vijay Machiraju, *Web services concepts, architectures and applications* (Springer Verlag 2004).

[ATSGB05] F Akal, C Türker, H Schek, T Grabs and Y Breitbart, Fine-grained lazy replication with strict freshness and correctness guarantees, *In Proc of International Conference on Very Large Data Bases (VLDB).* 2005: 31

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman, Concurrency control and recovery in database systems (Addison-Wesley, 1987).

[CORBA] Common Object Request Broker Architecture,

http://www.omg.org/technology/documents/spec_catalog.htm

[CRR96] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi, "Deferred Updates and Data Placement in Distributed Databases." *In Int. Conf. on Data Engineering, 1996, IEEE Computer Society* (1996) 469--476

[DCOM] Distributed Component Object Model,

http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-01.txt

[DS04] K Daudjee and K Salem, Lazy database replication with ordering guarantees. *In Proc of International Conference on Data Engineering (ICDE* 2004): 424-435

[DS05] Khuzaima Daudjee, Kenneth Salem, A Pure Lazy Technique for Scalable Transaction Processing in Replicated Databases. *In Proc of 11th International Conference on Parallel and Distributed Systems (ICPADS 2005)*, 20-22 July 2005, Fuduoka, Japan: 802-808

[ebXML] Electronic Business using eXtensible Markup Language,

http://www.ebxml.org/specdrafts/ebXML_TA_v0.9.pdf

[FM82] M. J. Fischer and A. Michael. Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network. *In Proc ACM PODS*, 1982: 70-75.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha, The danger of replication and a solution, *In Proc of ACM Special Interest Group on Management Of Data (SIGMOD)*, pages 173-182, 1996.

[GJ81] J. Gray et al., "A Straw-Man Analysis of the Probabilty of Waiting and Deadlocks in a Database System," IBM Research Report, 1981.

[GKO81] J. Gray, P. Homan, H. F. Korth, and R. L. Obermarck, 1981. A straw man analysis of theprobability of waiting and deadlock in a database system. Tech. Rep. RJ 3066, IBM Research Laboratory, San Jose, Calif.

[HSAE03] JoAnne Holliday, Robert C. Steinke, Divyakant Agrawal, Amr El Abbadi: Epidemic Algorithms for Replicated Databases. IEEE Trans. Knowl. Data Eng. 15(5): 1218-1238 (2003)

[KA98] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. *In Proc of International Conference on Distributed Computing Systems (ICDCS)* 1998: 156-163

[KA00] B. Kemme and G. Alonso, Don't be lazy, be consistent: postgres-R, a new way to implement database replication, *In Proc of International Conference on Very Large Data Bases (VLDB)* 2000: 134-143.

[NT88] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed systems. In *Proc. of the ACM Symp. on Principles of Distributed Computing (PODC)*, pages 248–262, Toronto, Canada, August 1988.

[S84] Schroeder, M.D, Experience with grapevine. *ACM Transactions on Computer Systems (TOCS)* 2, 1984: 3-23.

[SLK04] C. Sun, Y. Lin and B. Kemme, Comparison of UDDI registry replication strategies, *In Proc of IEEE International Conference on Web Services (ICWS)* 2004: 218-225

[SOAP] Specification: Simple Object Access Protocol, Version 1.2. http://www.w3.org/TR/soap12

[SS83] F. B. Schneider and R. D. Schlichting. *Fail-stop processors: an approach to designing fault-tolerant computing systems*. TOCS, 1(3):222--238, 1983.

[TDPSTW94] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welsh, Session guarantees for weakly consistent replicated data. *In Proc of Conference on Parallel and Distributed Information Systems (PDIS)* 1994: 140-149

[TG98] Tarafdar, Vijay K. Garg, Happened before is the wrong model for potential causality. TR-PDS-1998-006, UTA

[UDDI] Speciation: Universal Description, Discovery and Integration (UDDI), Version 3.0.1.
http://uddi.org/pubs/uddi-v3.0.1-20031014.pdf

[WSDL] Specification: Web Services Description Language (WSDL).
http://www.w3.org/2002/ws/desc/.

[X84] Xerox Corporation. Clearinghouse Protocol, XSIS 078404. Stamford, Connecticut, 1984.